8.1 Andmevahetus. Lugemine veebist. Failid

Seni oleme programmides vajalikud andmed kirjutanud programmi sisse või küsinud neid kasutajalt. Reaktsiooni oleme väljastanud ekraanile. Tegelikult võib ette tulla ka olukordi, kui algandmed tuleb hankida näiteks internetist ja tulemuse peaks salvestama hoopis faili. Käesolevas osas vaatleme andmevahetuse mõningaid võimalusi.

LUGEMINE VEEBIST

Võimalik, et meid huvitavad andmed on veebis kättesaadavad. Püüame sellise teksti kätte saada. Veebist lugemine pole eriti raske – tuleb kasutada käsku *urlopen*, mis on vaja eelnevalt importida moodulist *urllib.request*.

Olgu meid huvitav tekst järgneval aadressil https://kodu.ut.ee/~marinai/maa.txt .

Püüame teksti sealt siis otse kätte saada.

```
from urllib.request import urlopen

vastus = urlopen("https://kodu.ut.ee/~marinai/maa.txt")

baidid = vastus.read()
# veebist lugemisel annab käsk read() meile tavalise sõne asemel
hunniku baite,
# mis on vaja veel sõneks "dekodeerida"
tekst = baidid.decode()
vastus.close()

print(tekst)
```

Tekstides võib peituda salajasi teateid. Näiteks võime teada saada, mis tuleb täita lastega.

Ülesanne

Mis väljastatakse ekraanile, kui eelmises programmis rida

```
print(tekst)
```

asendatakse reaga

```
print(tekst[10].upper() + tekst[2] + tekst[-2])
```

Arutame nüüd, mida tekst[10].upper() + tekst[2] + tekst[-2] täpsemalt tähendab. Muutuja *tekst* väärtus on sõnetüüpi. Sõne koosneb sümbolitest (märkidest), seejuures loetakse eraldi sümboliks ka näiteks tühikut, reavahetust jms.

Sõnest saab sümboleid ühekaupa kätte indeksite abil. Erinevalt tavaelust on sümbolid sõnes nummerdatud nii, et tavamõttes esimene on indeksiga 0, teine indeksiga 1 jne. Negatiivse indeksi korral võetakse sümboleid alates sõne lõpust, nii on viimane sümbol indeksiga -1, eelviimane -2 jne.

Nii annab tekst[10] meile 11. sümboli ja funktsioon upper() muudab selle suurtäheks. Sümbol tekst[2] on tavamõttes kolmas sümbol ja tekst[-2] eelviimane.

Meenutame ülesannet, kus isikukoodi järgi tuli teada saada sünnikuupäev. Isikukoodi 4. ja 5. sümbol näitavad kuud, 6. ja 7. sümbol näitavad kuupäeva. Arvestades, et Python (ja paljud teised keeled) alustavad indeksite loendamist nullist, saame näiteks sellise programmi.

```
isikukood = input("Teie isikukood: ")
print("Teie sünnipäev on " + isikukood[5] + isikukood[6] + "." +
isikukood[3] + isikukood[4])
```

Selliseid andmestruktuure, mida saab käsitada elementide kaupa, on Pythonis mitmeid. Üheks selliseks on *list*, mida märgitakse nurksulgudega ja kus elemendid eraldatakse komadega. Omistame muutujale *a* väärtuseks listi, milles on arvud 1, 4, 5 ja -27.

```
a = [1, 4, 5, -27]
```

Listi *a* elemendi indeksiga 2 väärtus on 5.

Kõik listi elemendid saaksime ükshaaval ekraanile kuvada *while*-tsükli abil. Funktsioon *len* annab meile listi pikkuse ehk elementide arvu. Listi *a* pikkus on 4.

```
i = 0
while i < len(a):
    print(a[i])
    i += 1
```

Tegelikult saaksime listiga tegutseda *for*-tsükli abil, aga seda me selles kursuses ei tee.

FAILID. FAILIDE LUGEMINE REAKAUPA

Õpime ära ühe viisi, kuidas tekstifaile lugeda. Alustuseks võiks Thonnyga koostada ja salvestada tekstifaili nimega *andmed.txt*, mille esimesel real on inimese nimi, teisel real vanus (täisarvuna) ning kolmandal real meiliaadress (lihtsuse mõttes täpitähti ei kasuta).

NB! Loetav fail peab olema *plain-text* kujul (laiendiga *.txt*), Wordi fail (laiendiga *.doc* vms) näiteks ei sobi. Seepärast soovitamegi kasutada faili loomiseks Thonnyt.

Looge **tekstifailiga samasse kausta** järgnev programm.

```
f = open("andmed.txt")

nimi = f.readline()
vanus = f.readline()
aadress = f.readline()

print("Nimi:", nimi)
print("Vanus:", vanus, "aastat")
print("Aadress:", aadress)

f.close()
```

Vaatame nüüd selle programmi keerulisemaid ridu ükshaaval.

```
f = open("andmed.txt")
```

- Käsk *open* otsib failisüsteemist üles soovitud faili ja tagastab viite sellele (antud näites salvestasime selle viite muutujasse *f*, mis on levinud nimi failide tähistamiseks). Kui tahate avada faili samast kaustast, kus asub programm, siis piisab vaid failinimest koos laiendiga: f = open('andmed.txt').

```
nimi = f.readline()
```

- Rida nimi = f.readline() loeb failist ühe rea, milleks on meie näites isiku nimi, ning annab selle väärtuse sõnena edasi muutujale *nimi*. Järgmisel korral sama käsku kasutades loetakse järgmine rida. Meie näites loetakse failist järgmisena vanus.

```
f.close()
```

- Käsk f.close() ütleb failisüsteemile, et me oleme selle faili kasutamise lõpetanud.

Kui seda programmi katsetada, siis võib märgata, et väljundis tekib iga sisestatud andmejupi järele üks üleliigne tühi rida. Põhjus on selles, et failist lugedes jäetakse iga rea lõppu alles ka failist pärinev reavahetuse sümbol. Nimelt tähistatakse tekstifailides reavahetust sarnaselt Pythoniga: kui Pythoni sõne sisse kirjutame sümboli *n*, siis tähendab see Pythoni jaoks reavahetust. Iga kord kui tekstiredigeerijas järgmise rea ette võtate, sisestab programm teie eest faili reavahetuse märgi. See reavahetuse märk jäetakse alles, kui nüüd failist teksti loeme.

NB! Kui Python ütleb (Windowsi arvutis), et ei leia faili, aga olete veendunud, et fail on õiges kaustas olemas, siis tuleks kontrollida ega failinimele pole saanud eksikombel kaks faililaiendit. Segadust võib tekitada asjaolu, et operatsioonisüsteem varjab vaikimisi teatud faililaiendid. Kõige kindlam on muuta Windowsi seadeid nii, et alati näidataks kõiki faililaiendeid. Selleks tuleks mõnes kaustas valida menüüribalt *Vaade* ja seal panna "linnuke" punkti *Failinimede laiendid* ette (erinevates Windowsi versioonides võib see olla erinev).

NB! Kui proovida lugeda sisse täpitähtedega teksti, siis võib juhtuda, et saadakse veateade *UnicodeDecodeError*. Sel juhul tuleb *open* käsu rakendamisel öelda, millises kodeeringus on tekst, nt open('andmed.txt', encoding='UTF-8'). *UTF-8* asemel võib proovida ka *cp1257*.

FAILI SISU LUGEMINE ÜHEKORRAGA

Koostage veel üks mitmerealine, suvalise sisuga tekstifail ning salvestage see nimega *tekst.txt*. Seejärel käivitage (samas kaustas) järgmine näiteprogramm:

```
f = open('tekst.txt')
faili_sisu = f.read()
print(faili_sisu)
f.close()
```

Siin kasutasime meetodi *readline* asemel meetodit *read*, mis luges ühte sõne-tüüpi muutujasse sisse **kogu** faili sisu.

FAILIDE KIRJUTAMINE

Järgmine programm demonstreerib andmete kirjutamist tekstifaili:

```
nimi = input("Palun sisesta oma nimi: ")
vanus = input("vanus: ")
aadress = input("aadress: ")

f = open("andmed2.txt", "w")
f.write(nimi + "\n")
f.write(vanus + "\n")
f.write(aadress + "\n")
f.close()
```

Faili kirjutamiseks tuleb funktsioonile *open* anda ka teine argument väärtusega "w" (nagu *write* ehk otsetõlkes *kirjuta*). Kui sellise nimega fail juba eksisteerib, siis open(..., "w") teeb selle tühjaks. Erinevalt funktsioonist *print* ei tekita funktsioon *write* automaatselt reavahetusi. Selleks, et saada eri andmeid eri ridadele, lisasime reavahetuse sümboli käsitsi.

8.2 Suurem programminäide

Oleme eelnevalt tutvunud mitmesuguste võimalustega, mida programmide koostamisel saab kasutada. Programmid ei ole seni olnud väga pikad. Nüüd vaatleme veidi pikemat programmi, kus on nii varasemast tuttavaid asju, aga ka mõned sellised, millest veel juttu pole olnud.

Selles peatükis vaatleme ja täiendame ühte suuremat praktilist koodinäidet. Kas oskate alloleva koodi kohta vastata järgmistele küsimustele?

- Mis programmiga on tegemist?
- Milliseid andmestruktuure siin kasutatakse?
- Kuidas võivad sellises programmis kasuks tulla regulaaravaldised?

Programmi edukaks käivitamiseks on vaja ka tekstifaili:
https://courses.cs.ut.ee/MTAT.TK.006/2018_fall/uploads/Main/sonastik.txt

Sõnastiku fail on veidi modifitseeritud variant Eesti Keele Instituudi poolt jagatavast failist.

```
# -- PROGRAMMI 1. OSA - FUNKTSIOON TUTVUSTUS() --

def tutvustus():
    print("Inglise-eesti sõnastik (" + str(len(inglise_sõnad)) + "
sõna)")
    print("'v' - välju")
    print("'a' - abi")


# -- PROGRAMMI 2. OSA - FAILI SISSELUGEMINE --

f = open('sonastik.txt', encoding='UTF-8')

inglise_sõnad = []
eesti_sõnad = []
rida = f.readline()

while rida != '':
    sõnad = rida.split('\t')
    inglise_sõna = sõnad[0].strip().lower()
    eesti_sõna = sõnad[1].strip().lower()

    inglise_sõnad.append(inglise_sõna)
    eesti_sõnad.append(eesti_sõna)

    rida = f.readline()

f.close()



# -- PROGRAMMI 3. OSA - FUNKTSIOONI RAKENDAMINE JA OTSIMINE --
```

```
tutvustus()

while True:
    otsing = input('Sisesta otsitav ingliskeelne sõna: ')

    otsing = otsing.lower()

    if otsing == 'v':
        break
    elif otsing == 'a':
        tutvustus()
    elif otsing in inglise_sõnad:
        print('Inglise keeles: ' + otsing)
        otsitava_sõna_indeks = inglise_sõnad.index(otsing)
        print('Eesti keeles: ' + eesti_sõnad[otsitava_sõna_indeks])
    else:
        print('Sellist sõna pole!')
```

PROGRAMMIKOOD

Järgnevalt on esitatud sama programmikood koos kommentaaridega, et täpsemalt mõista, mida iga programmirida teeb:

```
# -- PROGRAMMI 1. OSA - FUNKTSIOON TUTVUSTUS() --

# Defineerime funktsiooni, mis trükib ekraanile programmi tutvustuse
ja sõnastikus olevate sõnade arvu
def tutvustus():
    print("Inglise-eesti sõnastik (" + str(len(inglise_sõnad)) + "
sõna)")
    print("'v' - välju")
    print("'a' - abi")


# -- PROGRAMMI 2. OSA - FAILI SISSELUGEMINE --

# Avame faili 'sonastik.txt', mis on kodeeringus UTF-8
f = open('sonastik.txt', encoding='UTF-8')

# Kõik sõnad sorteerime listidesse vastavalt keelele
inglise_sõnad = []
eesti_sõnad = []

# Loeme failist esimese rea
rida = f.readline()

# Loeme failist ridu nii kaua, kuni neid jätkub
while rida != '':

    # Jagame rea tükkideks tabulaatori (TAB) kohalt (Pythonis '\t')
    sõnad = rida.split('\t')
```

```
    # Rea esimene pool (enne tabulaatorit) on ingliskeelne sõna
    # strip() eemaldab tühikud ja reavahetused sõne algusest ja
lõpust
    inglise_sõna = sõnad[0].strip().lower()
    # Rea teine pool (pärast tabulaatorit) on eestikeelne sõna
    eesti_sõna = sõnad[1].strip().lower()

    # Lisame ingliskeelse ja eestikeelse sõna vastavatesse
listidesse
    inglise_sõnad.append(inglise_sõna)
    eesti_sõnad.append(eesti_sõna)

    # Loeme sisse järgmise rea, pärast seda hakkab tsükkel otsast
peale
    rida = f.readline()

# Sulgeme faili
f.close()


# -- PROGRAMMI 3. OSA - FUNKTSIOONI RAKENDAMINE JA OTSIMINE --

# Rakendame funktsiooni tutvustus()
tutvustus()

# Küsime kasutajalt sõne, otsime sellele vaste ning väljastame selle
# Seda teeme lõpmatu tsükliga, mida katkestame õiges kohas käsuga
'break'
while True:

    # Küsime kasutajalt sõna
    otsing = input('Sisesta otsitav ingliskeelne sõna: ')

    # Kui kasutaja sisestatud sõnas on suurtähti, siis nüüd teeme
need väikseks
    otsing = otsing.lower()

    # Kui kasutaja sisestab 'v', siis lõpetame tsükli
    if otsing == 'v':
        # Katkestame tsükli tegevuse, programmi täitmist jätkatakse
tsükli lõpust
        break

    # Kui kasutaja sisestab 'a', siis rakendame funktsiooni
tutvustus()
    elif otsing == 'a':
        tutvustus()

    # Kui kasutaja sisestatud sõna on listis 'inglise_sõnad', siis
otsime sõnale vaste ning väljastame selle
    elif otsing in inglise_sõnad:
        print('Inglise keeles: ' + otsing)
        # Leiame, mis indeksiga kohal on 'otsing' listis
'inglise_sõnad
```

```
        otsitava_sõna_indeks = inglise_sõnad.index(otsing)
        # Vaste on samal kohal listis eesti_sõnad
        print('Eesti keeles: ' + eesti_sõnad[otsitava_sõna_indeks])

    # Kui ükski eelnevatest tingimustest ei ole tõene, siis
sõnastikus sellist sõna ei leidu
    else:
        print('Sellist sõna pole!')
```

PROGRAMMI LÜHIKIRJELDUS

Nagu programmist on võimalik välja lugeda, on tegemist inglise-eesti sõnastikuga. Programmi käivitamisel loetakse sõnastiku read tekstifailist mällu ning seejärel saab teha sõnadele päringuid. Sõnastikuna on kasutatud tekstifaili, mis on UTF-8 kodeeringus (kättesaadav ülaltpoolt).

Programmi käivitamisel avaneb vaade:

```
>>>
Inglise-eesti sõnastik (88898 sõna)
'v' - välju
'a' - abi
Sisesta otsitav ingliskeelne sõna:
```

Esmalt loeb programm mälusse sõnastiku. Lõik sõnastikust näeb välja niimoodi:

computer software	tarkvara, arvutitarkvara

Nagu näha, eristuvad eestikeelsed sõnaseletused ingliskeelsest väljendist sellepoolest, et need on eraldatud tabulaatori ehk taandemärgiga (kirjutatakse Pythonis "\t" või '\t'). See tähendab, et iga rida tähistab uut sissekannet sõnastikus. Lugedes faili reakaupa sisse, saame iga rea puhul eristada ingliskeelsed märksõnad ja eestikeelsed vasted nende vahel oleva taande abil. Selleks kasutame funktsiooni *split* niimoodi: `sõnad = rida.split('\t')`. Siin muutuja *rida* on sõne, mis sisaldab ühte rida failist. Võime mõelda, et *split* tükeldab antud sõne tükkideks nii, et esimene tükk on see, mis oli enne tabulaatorit ja teine tükk see, mis tuli pärast tabulaatorit.

Seega sisaldab esimene tükk ingliskeelset mõistet ja teine tükk peaks olema eestikeelne vaste sellele. Kui seda teame, siis saame eesti- ja ingliskeelsed mõisted sorteerida. Mõistete hoidmiseks kasutame andmestruktuuri, mille nimi on *list* (seda sai ka eelmise nädala materjalides põgusalt mainitud). List erineb teistest meile tuntud andmestruktuuridest nagu arv (*int* või *float*) ja sõne (*str*) selle poolest, et sinna saab salvestada mitu väärtust (infokildu) korraga. See on justkui paberite virn, kus iga listis salvestatud väärtus on üks paber. Igal paberil ehk väärtusel on ka oma indeks ehk järjekorranumber (nagu ikka programmeerimises kombeks, algab loendamine 0-st). Listis olevaid väärtusi kutsutakse selle listi elementideks ning need kirjutatakse nurksulgude ("[" ja "]") vahele. Tühja listi saame tekitada näiteks sellise omistuslausega `eesti_sõnad = []`: kuna nurksulud on tühjad, siis pole selles listis ühtegi elementi.

Saladuskatte all võime öelda, et funktsioon *split*, millega rida tükeldasime, tagastab samuti listi. Ütlesime, et esimene tükk sisaldab ingliskeelset mõistet ning teine eestikeelset, kuid tegelikult need tükid ongi listi elemendid. Seega tagastatakse *split*i toimel meile sõnedest koosnev list, mille esimeseks elemendiks on ingliskeelne mõiste ning teiseks elemendiks eestikeelne vaste.

Nüüd salvestame ingliskeelse mõiste muutujasse *inglise_sõna* ning selle käigus kasutame ka sõnemeetodeid *strip* ja *lower*. Esimene neist eemaldab sõne otstest kõik tühikud ja reavahetused, teine muudab kõik tähed sõnes väiketähtedeks. Samamoodi toimime ka eestikeelse vastega.

Listi lõppu saab elemente lisada käsuga *append*, näiteks `eesti_sõnad.append(eesti_sõna)` lisab listi nimega *eesti_sõnad* muutuja *eesti_sõna* väärtuse, milleks on loomulikult sõne. Oluline on, et *append* lisab elemendi just listi lõppu, mitte suvalisse kohta.

Viimase tegevusena tsüklis loeme sisse uue rea failist: `rida = f.readline()`. Sellega saab muutuja *rida* endale väärtuseks sõne, mis sisaldab järgmist rida antud failist. Kui tekstifailis järgmist rida ei ole (fail on lõppenud), siis saab eelneva käsu abil muutuja *rida* väärtuseks tühja sõne ehk "" või ''. Sellest tulebki tsükli jätkamistingimus: kui *rida* väärtus ei ole tühi sõne, siis tsükliga jätkatakse ehk loetakse järgmine rida.

Niimoodi saamegi kogu faili loetud ja lõpuks on meil listid *eesti_sõnad* ja *inglise_sõnad*, mis sisaldavad vastavaid mõisteid. Tuleks märgata, et listides on elemendid täpselt samas järjekorras - see tähendab, et eestikeelsele mõistele, mis asub listis *eesti_sõnad* indeksil 1, vastab ingliskeelne mõiste, mis on listis *inglise_sõnad* indeksiga 1. Seega on kokkusobivate mõistete indeksid võrdsed.

Nüüd algab programmiosa, mis tegeleb ka kasutajaga suhtlemisega.

Esmalt väljastatakse eelnevalt defineeritud funktsiooni rakendamise teel sissejuhatav tekst. Seejärel küsitakse kasutajalt otsitav ingliskeelne sõna ning teisendatakse selle tähed väiketähtedeks. Siis kontrollitakse, kas kasutaja sisestas tähe "v": kui ta seda tegi, siis väljutakse tsüklist käsuga *break*. Kui kasutaja sisestatud sõne oli midagi muud, siis kontrollitakse, kas tegemist oli tähega "a" (sel juhul väljastatakse jällegi sissejuhatav abitekst).

Kui kasutaja ei sisestanud "a" ega "v", siis kontrollime, kas ta sisestus on üks listi *inglise_sõnad* elementidest. Seda kontrollitakse võtmesõnaga *in*: `otsing in inglise_sõnad` tagastab *True*, kui *inglise_sõnad* sisaldab elementi, mille väärtus on võrdne muutuja *otsing* väärtusega, vastasel juhul tagastatakse *False*. Kui eelnev tingimus on täidetud, siis leidub meil sõnastikus see sõna, mida kasutaja soovis. Kui see tingimus täidetud ei ole, siis minnakse tingimuslause *else*-harru ning väljastatakse vastav teade.

Oluline on märgata, et tingimuslauses täidetakse alati täpselt üks haru. Kui üks tingimustest on tõene, siis täidetakse see haru, kui aga ükski tingimus tõene pole, siis täidetakse *else*-haru.

Soovitame programm ühe korra veel üle vaadata ja mõelda läbi, kas saite aru, kuidas kõik töötab.

8.3 Silmaring. Navigeerimine

NAVIGEERIMINE

Sõna navigeerimine tuleb ladina keelest, kus *navis* tähendab laeva ning *agere* tähendab juhtimist. (Allikas: https://www.britannica.com/technology/navigation-technology)

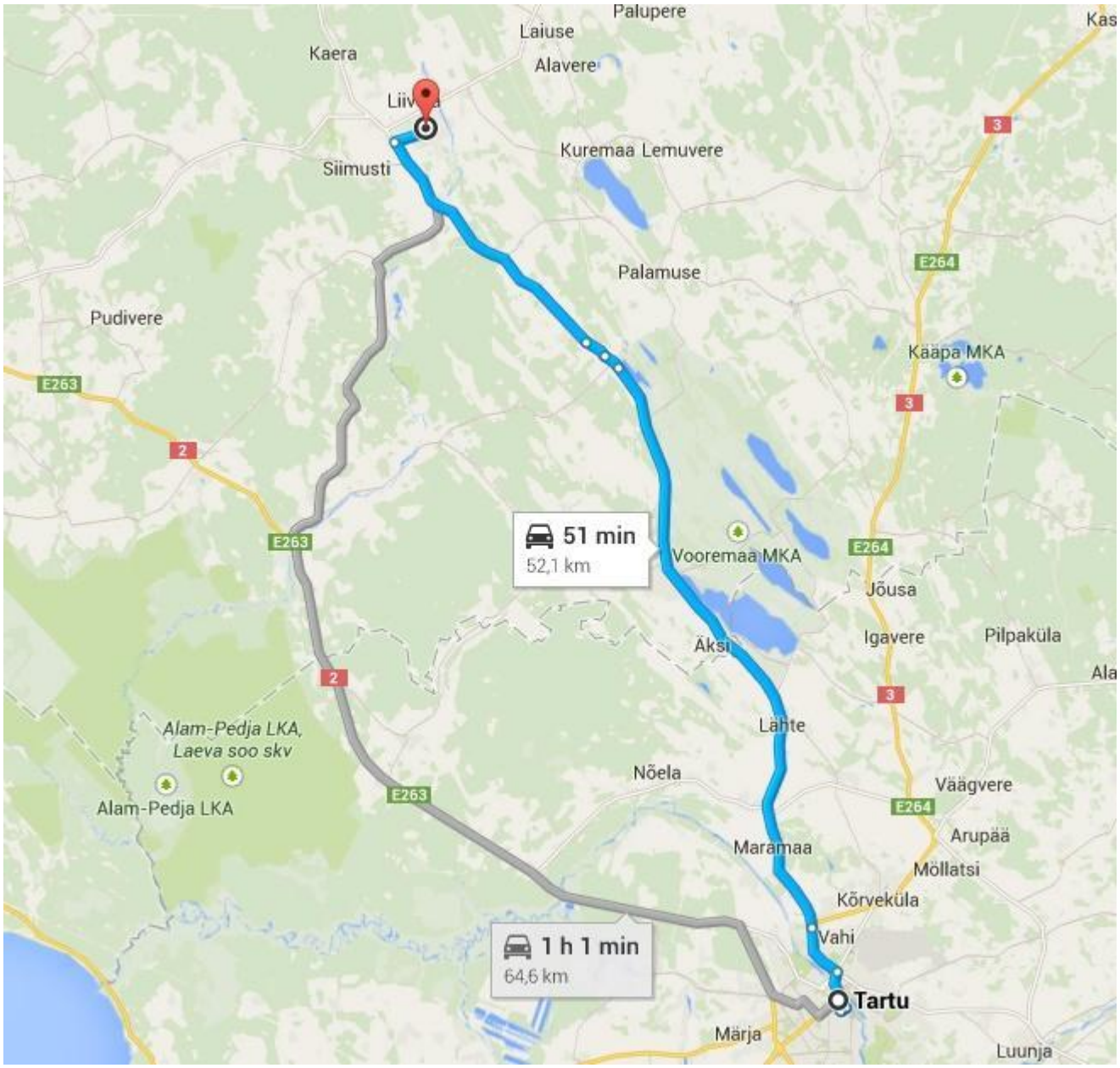
Nii vanasti kui ka nüüd peetakse navigeerimise all silmas sõiduki juhtimist kasutades navigatsioonivahendeid - vahendeid, mis aitavad määrata asukohta ja kurssi. Tänapäevased vahendid püüavad endisaegsetest rohkem abi pakkuda. Näiteks võivad nad pakkuda (mingis mõttes) sobivaimat marsruuti.

Nii mõnedki meist harjunud autoga sõitma nii, et (paber)kaarti kaasas ei olegi. Meie sõitu suunab armatuurlaual asuv GPS seade või hoopis telefon. Oleme harjunud seda seadet usaldama - ta näitab meile teed sinna, kuhu minna tahame. Aga kuidas ta selle tee meie jaoks leiab? Mismoodi ta meid õigeks ajaks õigesse kohta juhatab? Kas me ikka julgeme seda seadet enda juhtimisel usaldada? Kuidas see kõik programmeeritud on?

Viimasel ajal on igasuguseid navigeerimisrakendusi üsna palju täiustatud. Näiteks telefonirakendus *Waze* toimib ainult kasutajate tagasiside najal. Kui keegi sõidab, märgib ta kaardile erinevaid ohte ja õnnetusi käsitsi või kui mitmed rakenduse kasutajad läbivad teatud teelõike aeglaselt, oskab see rakendus uue tee arvutada, vältides aeglasemaid teelõike. Antud peatüki eesmärk ongi põgusalt vaadata, kuidas navigeerimisrakendused erinevaid teekondi valivad.

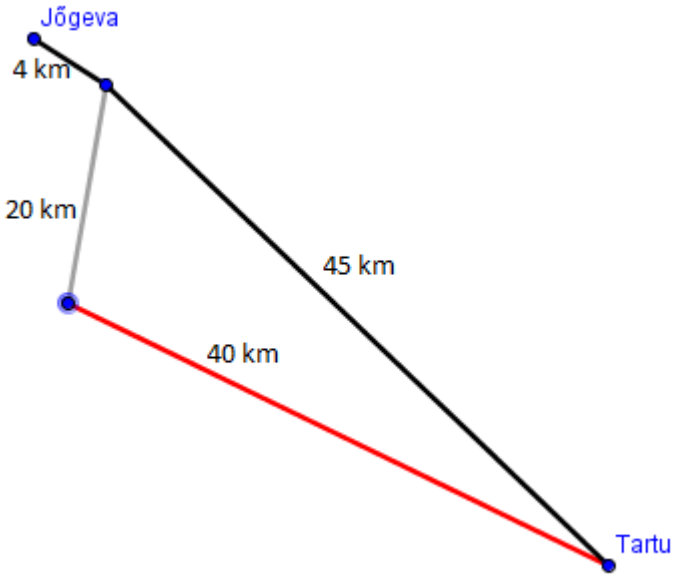
VAATAME KAARTI

Soovime sõita näiteks Tartust Jõgevale. Selleks pakub navigeerimisrakendus meile kaht eri pikkusega teelõiku, millest ta on ühe välja valinud, kuna see on tema arvates kiirem. Kuidas ta teab, kumb kiirem on? Võtame appi matemaatika.



Kirjeldame olukorda joonise abil. Lihtsustame reaalset olukorda järgmiselt. Igal teelõigul on määratud piirkiirus:

- punane - 100 km/h
- must - 70 km/h
- hall - 50 km/h



Hetkel on võimalik Tartust Jõgevale minna kaht teed pidi.

1. Ühe pikkuseks on 45 km + 4 km = 49 km,
2. teise tee pikkuseks on 40 km + 20 km + 4 km = 64 km.

Teine tee on pealtnäha palju pikem, kuid arvesse tuleb võtta ka teeolusid. Esimese tee puhul on auto keskmiseks kiiruseks 70 km/h. Teise tee esimesel osal on keskmiseks kiiruseks 100 km/h, teisel osal 50 km/h ning kolmandal 70 km/h.

Kumb tee on kiirem Tartust Jõgevale sõitmiseks?

Esiteks leiame iga teelõigu läbimiseks kuluva aja.

- Kiiruse leidmiseks on olemas valem:

$$v = \frac{s}{t}$$

, kus v - kiirus, s - teepikkus, t - aeg

- Antud valemist tuletame kuluva aja:

$$t = \frac{s}{v}$$

Liidame mõlema teekonna teelõikude ajad kokku ja võrdleme neid:

Lahendus:

- I teelõigu läbimiseks kulub

$$\frac{45\ km + 4\ km}{70\ km/h} = 0{,}7\ h = 42\ min$$

- II teelõigu läbimiseks kulub

$$\frac{40\ km}{100\ km/h} + \frac{20\ km}{50\ km/h} + \frac{4\ km}{70\ km/h} = 0{,}4h + 0{,}4h + 0{,}05h = 0{,}85\ h = 51\ min$$

Seega tuleks kiiruse huvides valida just esimene marsruut. Analoogiliselt leiab arvuti kiireima marsruudi ka siis, kui võimalikke teekondi on tuhandeid või isegi miljoneid.

NÄIDE I

Järgmistes programminäidetes kasutame järjendeid ehk liste (andmetüüp, milles saab korraga hoida mitut väärtust) ning *for*-tsüklit, mida me varasemalt käsitlenud pole. Samas leidub ka juba tuttavat nagu funktsioonid ja *while*-tsükkel. Antud programmid on lihtsalt uudistamiseks ning testiküsimusi nende kohta ei ole. Püüdke siiski konteksti ja kommentaaride järgi aru saada, mida tehakse ja miks seda vaja on.

Esiteks koostame kahekordsed järjendid mõlema teekonna jaoks. Järjend koosneb teejuppide järjenditest, kus esimesel kohal on teepikkus kilomeetrites ning teisel kohal lubatud kiirus kilomeetrites tunnis. Näiteks tee, mis koosneb kolmest teelõigust (muutuja `rada1`) on esimese teelõigu pikkus 40 km ja lubatud kiirus 100 km/h (järjendi esimene element `[40, 100]`), teise teelõigu pikkus 20 km ja 50 km/h (järjendi teine element [20, 50]) jne.

```
rada1 = [[40, 100], [20, 50], [4, 70]]
rada2 = [[45, 70], [4, 70]]
```

Defineerime funktsiooni aja arvutamiseks kiiruse ja teepikkuse järgi.

```
def aeg (s, v):                          # Argumendid: teepikkus ja
kiirus
    if s > 0 and v > 0:                  # Teepikkus ja kiirus on
positiivsed suurused
        t = float(s) / float(v) * 60     # Teisendame ujukomarvudeks ja
minutitesse
        return t
    else:
        print ("Teepikkus ja aeg peavad olema positiivsed arvud!")
```

Leiame mõlema teekonna läbimiseks kuluva aja. Esimese teekonna läbimiseks kuluva aja arvutamine (kasutame *while*-tsüklit):

```
kuluv_aeg_1 = 0
i = 0
while i < len(rada1):                  # Kestab nii kaua kuni kõik
järjendi elemendid on läbitud
    x = aeg(rada1[i][0], rada1[i][1]) # Kasutades varasemalt
defineeritud alamprogrammi 'aeg', leiame igal teelõigul kuluva aja
    kuluv_aeg_1 = kuluv_aeg_1 + x     # Liidame teelõikude ajad
kokku
    i = i + 1
kuluv_aeg_1 = int(kuluv_aeg_1)         # Teisendame täisarvuks
```

Leiame sarnaselt ka teise teekonna jaoks kuluva aja (seekord kasutame vahelduseks *for*-tsüklit, huvi korral proovige seda ka ise mõnes programmis kasutada).

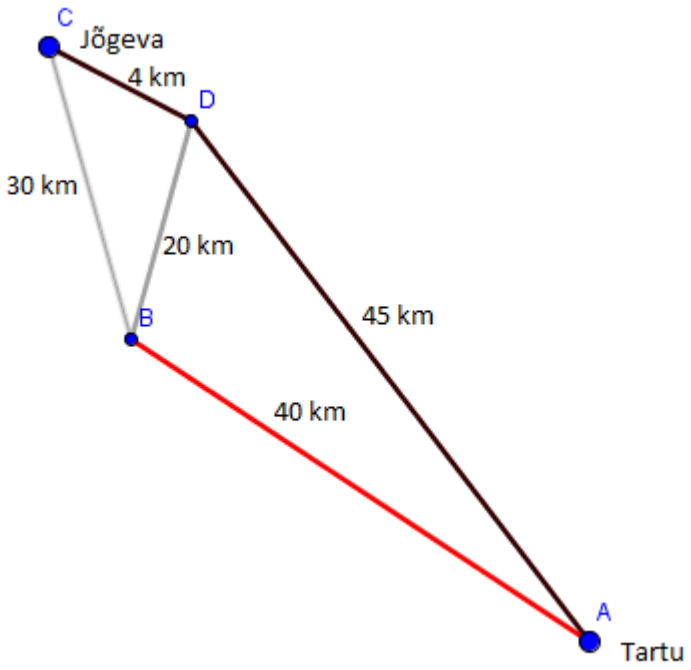
```
kuluv_aeg_2 = 0
for i in range(len(rada2)):
    x = aeg (rada2[i][0], rada2[i][1])
    kuluv_aeg_2 = kuluv_aeg_2 + x     # Liidame ajad kokku
kuluv_aeg_2 = int(kuluv_aeg_2)         # Teisendame täisarvuks
```

Nüüd võrdleme aegasid. Väljastame ekraanile, kumb teekond on kiirem ning kui palju selle läbimiseks aega kulub.

```
if kuluv_aeg_1 == kuluv_aeg_2:
    print("Mõlemat teed pidi jõuab kohale sama kiiresti, aega kulub
" + str(kuluv_aeg_1) + " minutit.")
elif kuluv_aeg_1 < kuluv_aeg_2:
    print("Esimene tee on kiirem, aega kulub " + str(kuluv_aeg_1) +
" minutit.")
else:
    print("Teine tee on kiirem, aega kulub " + str(kuluv_aeg_2) + "
minutit.")
```

NÄIDE II

Mis juhtub, kui võimalikke teid tuleb juurde? Vaatleme järgmist teede joonist:



Ülesanne: Mitu erinevat võimalust on nüüd Tartust Jõgevale sõitmiseks?

- Tartu - B - Jõgeva
- Tartu - D - Jõgeva
- Tartu - B - D - Jõgeva
- Tartu - D - B - Jõgeva

Seega ühe tee lisamisega tekkis juurde kaks võimalust. Kõige kiirema tee leidmiseks kasutame esimeses näites selgitatud arvutuskäiku. Kõik erinevad teekonnad me juba leidsime, nüüd tuleb nende läbimiseks kuluvad ajad eraldi välja arvutada ning seejärel neid võrrelda.

Leia erinevate teekondade läbimiseks kuluvad ajad esimese näiteprogrammi järgi. Kas programmi saaks kuidagi “ilusamaks” teha?

Vihje: Defineeri veel üks funktsioon:

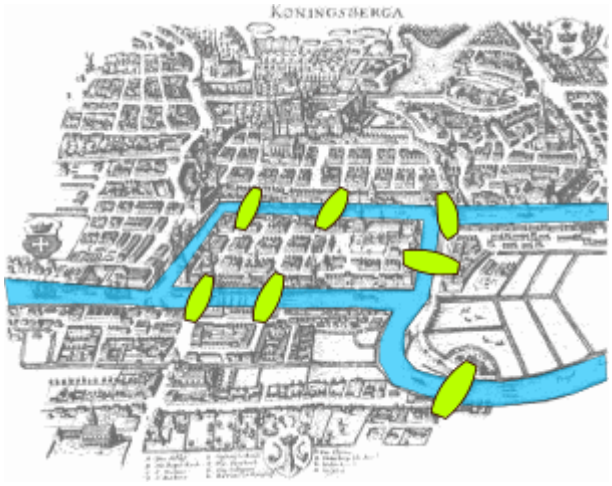
```
def kuluvaeg (rada):
    i = 0
    kuluv_aeg = 0
    while i < len(rada):
        x = aeg(rada[i][0], rada[i][1])
        kuluv_aeg = kuluv_aeg + x
        i = i + 1
    kuluv_aeg = int(kuluv_aeg)        # Teisendame täisarvuks
    return kuluv_aeg
```

Edasimõtlemiseks: kuidas küsida võimalikke teekondi kasutajalt?

KÖNIGSBERGI SILDADE PROBLEEM

Eestist mitte väga kaugel asub umbes Tallinna-suurune linn, mis aastasadu kandis nime Königsberg, aga pärast II maailmasõda sai nimeks Kaliningrad.

Vanasti oli Pregeli (Pregolja) jõel seitse silda.



Turistide seas kerkis küsimus: kas on võimalik üle kõigi sildade jalutada nii, et iga silda ületatakse täpselt üks kord ning jõutakse lõpuks alguspunkti tagasi?

Legendaarne matemaatik Leonhard Euler tõestas aastal 1736, et neid sildu ei olegi niiviisi võimalik läbida. Sellega pandi alus matemaatika harule, mille nimeks tänapäeval on graafiteooria. Graafiteooria on teadus, mis uurib graafe - skeeme-graafikuid, millel vaadeldakse objektide omavahelisi seoseid.

Selliseid lihtsamaid graafe leidub näiteks nuputamisülesannetes, kus tuleb ühe joonega kujundeid valmis joonistada (näiteks ühe joonega ümbrik). Selliseid mänge on ka nutiseadmetel ja arvutis mängimiseks. Graafiteooria abil saab tõestada näiteks huvitava fakti, et kui majal on üks välisuks, siis selles majas peab leiduma vähemalt üks ruum, kuhu viib paaritu arv uksi.

Tänapäeval on graafiteooria väga aktuaalne nii matemaatikas kui arvutiteaduses. Näiteks saab graafidega kujutada erinevaid suhtlusvõrgustikke, andmevooge, keerukaid kaarte ja palju muud. Kui tekib selle teema vastu suurem huvi, siis rohkem infot leiab näiteks siit.

Materjali koostasid Kerri Gertrud Vestberg ja Mari-Liis Jaansalu. Kohendatud kursuse korraldajate poolt.

8.4 Maalähedane lugu VIII

Rasmus tõesti ei kõõlunud tooliga, see lihtsalt lagunes ära. Kuna Lembitut hetkel polnud, kes kohe parandamise oleks ette võtnud, ütles Ada Rasmusele, et see tooli pööningule viiks. Rasmus ja Liisbeth läksidki pööningule ja tulid sealt tagasi alles mõne aja pärast, üks kenasti kinninööritud pakk kaasas. Need olid vanad kirjad ja postkaardid - veidi koltunud, aga suuresti ikka loetavad. Enamik kirju ja kaarte oli siia tallu saadetud. Mõned ka sellest talust saadetud, aga siis koos adressaadiga hiljem tagasi tulnud. Või sellest talust küll saadetud, aga saajat leidmata saatjale tagasi toodud.

Ada pühkis käed puhtaks ja tuli noorte juurde. See oli aja lugu, mis nüüd toa täitis. Kõige vanemad kirjad olid sellest ajast, kui Ada oli väike tüdruk olnud. Kõigi inimeste kohta oli Adal midagi öelda. Kuna ta kraamis välja ka vanad albumid ja rääkis üsna piltlikult, siis kujutasid Liisbeth ja Rasmus küllalt selgesti ette, kuidas need inimesed tulid ja läksid. Ada teadis saajatest-saatjatest peaaegu kõiki, Liisbeth tegelikult ainult paari. Rasmus jäi oma teadmistega nende vahele. Muidugi teadis Adat ja mäletas natuke Karli - need olid tema vaarvanemad. Samuti tundis ta nende lapsi Lembitut, Jaani ja Tiinat. Tiina oli Rasmuse vanaema. Lembit seega Rasmuse vanaonu. Eks ta oli ikka juttudes kuulnud teistest ka.

Inimeste kõrval köitsid noorte tähelepanu aadressid, postmargid ja templid. Imelik, kui palju erinevaid aadresse oli ühel samal talul olnud. Nad said Ada käest teada, mida tähendasid lühendid sjk, k/n, mida tähendas Feldpost, miks mõnedel kirjadel on margi asemel CA kirjutatud, millal oblastid olid olnud ja palju muud. Eks midagi ju koolist ka kuuldud, aga nüüd tundus see kuidagi päris olevat! Eestist saadetud markide hindu olid markades, sentides, kroonides ja kopikates. Markide peal oli erinevaid inimesi, vappe, ehitisi. Ühe 1932. aasta margi peal oli Tartu Ülikooli peahoone. Kirigi oli ka Tartust tulnud. Osa kirju oli aga tulnud väga kaugelt ja väga kaua. Nendest rääkides tundusid Ada silmad kuidagi läikivat ...

VAHELEPÕIGE Ada, Liisbeth ja Rasmus muidugi ei mõenud sellele, et aadressid on tähtsad ka programmeerimises. Nii saab teadaolevalt veebiaadressilt vajaliku info kätte. Igal arvutil võrgus on oma aadress. Igasugused andmed salvestatakse mällu - olulised on mäluaadressid. Aadresside kirjutamiseks on oma reeglid.

Veel oli huvitav vaadata, millega tekst kirjutatud oli. Paljud kirjad paistsid olema tindiga kirjutatud. Noored olid Palamuse muuseumis isegi sule ja tindiga kirjutada proovinud. Osad kirjad olid hariliku pliiatsiga kirjutatud. Paar kaarti aga tundusid olevat kirjutatud nagu tindi ja hariliku pliiatsiga korraga. Ega Liisbeth ja Rasmus nüüd Ada selgitusest küll päriselt aru ei saanud, et mis tindipliiats ja teha pliiatsi ots märjaks. Kes siis tintenpeni märjaks teeb!? Õnneks leidis Ada köögilaua sahtlist ühe tindipliiatsi üles. Ja oligi nii - muidu nagu harilik pliiats, aga kui pliiatsi ots märjaks teha, hakkas tindiga kirjutama!

Aeg läks kiiresti ja juba oligi õhtu käes. Köögist läbi minnes märkas Liisbeth, et telefoni juures oli pabertükk, millel olid kirjas koordinaadid - põhjalaius ja idapikkus. See oli selleks, et vajadusel väga täpselt oma kohast teada anda, mine tea, millal kiiresti abi vaja on. Talu asus külast natuke kaugemal ja täpset aadressi, tänava ja majanumbriga ju polnudki.

Lisamaterjale ja lisaülesandeid

Siia on kogutud selliste materjalide linke, mis avardavad teie teadmisi.

Julgesti võite linke juurde pakkuda!

- Kilpkonn joonistab rekursiivselt puid
- Veel üks suurem programm

Lisaülesanne: Pere sissetulek 2017

2018. aastal kehtestati uus ja keerulisem maksuvaba tulu arvestamise kord. See ülesanne on veel ajast, kui maksuvaba miinimum oli fikseeritud.

Defineerige funktsioon nimega *netopalk*, mis võtab argumendiks inimese brutopalga, arvutab sellest netopalga ja tagastab selle. Automaatse kontrollimise võimaldamiseks lepime kokku, et brutopalgast arvatakse maha vaid tulumaks (20%). Seda arvestatakse sellest osast palgast, mis jääb üle maksuvaba miinimumi (170€). Netopalk peab olema ümardatud 2 komakohani. **Ümardada tuleb juba funktsiooni sees.** Ümardamiseks saab kasutada funktsiooni *round*. Katsetage ise, kuidas funktsioon *round* töötab!

Kirjutage programm, mis küsib isa brutopalga, ema brutopalga ning alaealiste laste arvu, ja arvutab selle põhjal pere kuusissetuleku. Sisendandmed tuleb küsida sellises järjekorras, nagu neid ülesande tekstis mainitakse. Oletame, et iga alaealise lapse kohta makstakse toetust 50€ kuus. Kogu pere sissetulek tuleb kuvada ümardatuna 2 komakohani.

Näide programmi tööst:

```
>>> %Run pereSissetulek.py

  Sisestage isa palk: 2500
  Sisestage ema palk: 1750
  Sisestage laste arv: 2
  Pere kuusissetulek 3568.0

>>>
```

Selle ülesande lahenduse võib esitada *Moodle*'is ja saada automaatset tagasisidet, aga kohustuslik see ei ole.

Kui on huvi, siis võib teha programmi ka praeguse süsteemi kohta, aga sellel automaatkontrolli ei ole.