4.1 VIITAMINE

Viidad ja väärtused

Selles osas avame natuke sügavamalt erinevat tüüpi muutujate ja väärtuste olemust. Mis täpsemalt juhtub, kui muutujale antakse väärtus? Ja mis juhtub siis, kui seda väärtust tahta muuta või sama väärtust anda teisele muutujale?

Muutujale saab väärtuse omistada võrdusmärgi abil. Oleme seda teinud juba meie kursuste algusest peale. Näiteks `a = 6` toimel saab muutuja `a` väärtuseks `6`. Harilikult meile sellest teadmisest piisab. Siiski on huvitav ja mingites olukordades ka vajalik teada, et tegelikult salvestatakse `x = 6` toimel muutujasse `x` hoopis viit objektile, mis tähistab mälus arvu `6`.

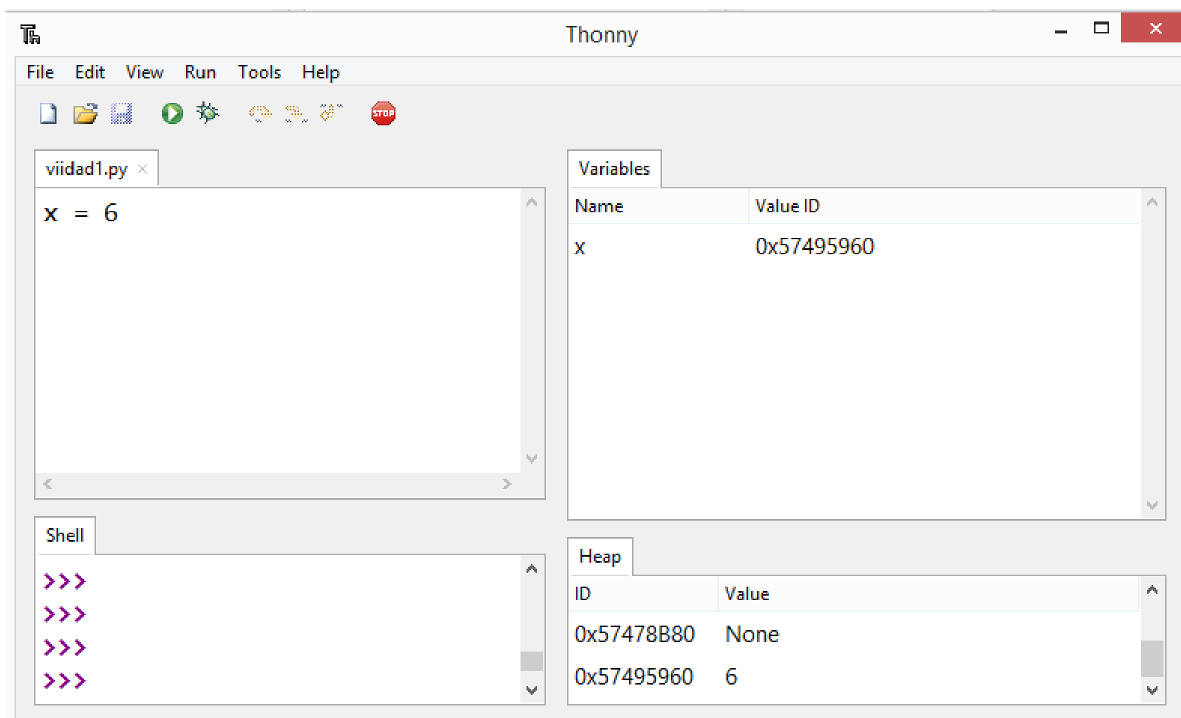
Seda viita saame me näha täisarvuna funktsiooni `id` abil.

```
x = 6
print(id(x))
```

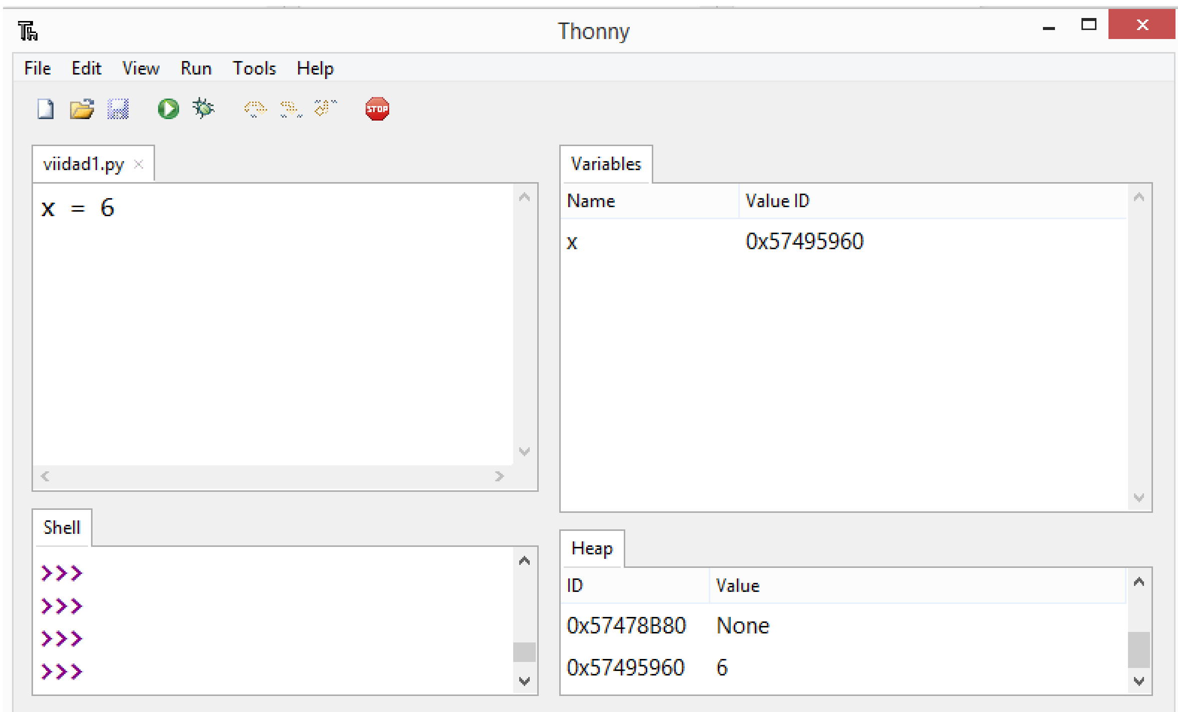
Väljastatakse arv `1464424800`, mis on unikaalne ja konstantne kogu objekti eluea jooksul.

Thonnyl on head võimalused väärtuste ja viitade jälgimiseks.

Kui avada *Variables*-aken (*View* --> *Variables*), siis näeme, et `x`-i väärtus on `6`.

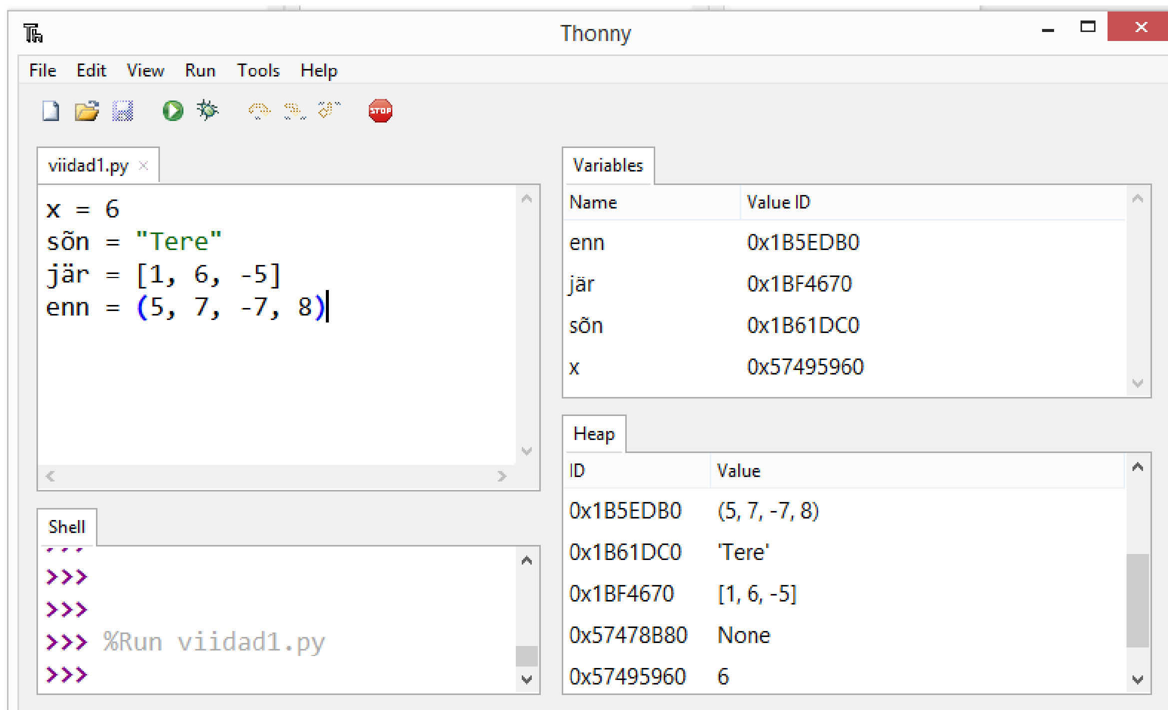


Kui aga avame ka *Heap*-akna (*View* --> *Heap*), siis näeme, et muutuja juures on kirjas hoopis viit ja viida juures väärtus. Natuke segadust võib tekitada see, et funktsioon `id` andis viidaks `1464424800`, aga Thonnys *Variables*- ja *Heap*-akendes paistab `0x57495960`. Tegelikult on tegemist siiski sama arvuga, `0x57495960` on lihtsalt kuueteistkümnendsüsteemis.



Samamoodi on viidad ja väärtused ka teiste andmetüüpidega. Võtame kasutusele näiteks veel ühe sõne, ühe järjendi ja ühe enniku.

```
x = 6
sõn = "Tere"
jär = [1, 6, -5]
enn = (5, 7, -7, 8)
```



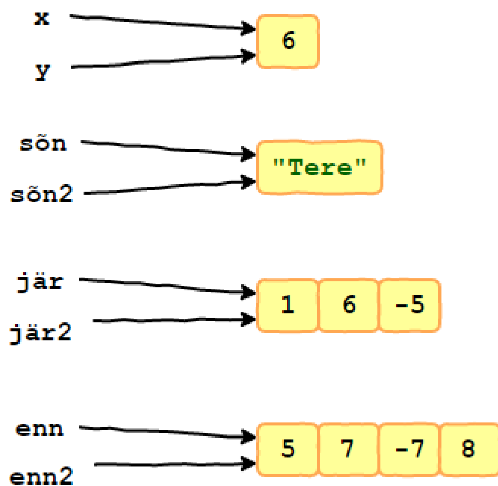
Näeme, et *Heap*-aknas, on rohkem asju, kui ainult muutujate viidad, aga teisi me praegu ei vaata.

Mitmel muutujal sama viit

Kui nüüd teha omistus `y = x`, siis muutujasse `y` salvestatakse sama viit ja uut täisarvu objekti ei looda. Samuti on ka teiste andmetüüpide puhul.

```
x = 6
y = x
sõn = "Tere"
sõn2 = sõn
jär = [1, 6, -5]
jär2 = jär
enn = (5, 7, -7, 8)
enn2 = enn
```

Jooniselt näeme, millised muutujad viitavad samale objektile.



Sama viidaga muutujaid võib veel rohkem olla.

```
jär3 = jär2
jär4 = jär2
```

Mis aga juhtub, kui näiteks anda mingitele uutele muutujatele kohe need samad väärtused?

```
x = 6
y = x
sõn = "Tere"
sõn2 = sõn
jär = [1, 6, -5]
jär2 = jär
enn = (5, 7, -7, 8)
enn2 = enn
z = 6
sõn_uus = "Tere"
jär_uus = [1, 6, -5]
enn_uus = (5, 7, -7, 8)
```

Näeme, et arvu ja sõne puhul on nüüd viit ikka sama, aga järjendi ja enniku puhul mitte. Sama näiteks ka tühja enniku puhul on erinevate muutujate korral viit sama.

```
tühi_ennik = ()
tühi_ennik2= ()
```

Järjendi korral aga ei juhtu seda kunagi. Räägime sellest lähemalt järgmise peatüki lõpus.

4.2 MUTEERIMINE

Me oleme oma kursuste jooksul ikka mingite muutujate väärtust muutnud. Juba termin *muutuja* isegi viitab muutumisele.

```
a = 5
a = 7
a += 4
```

Vaatame nüüd sedagi aspekti mõnevõrra sügavamalt.

```
a = 5
print(id(a))
a = 7
print(id(a))
a += 4
print(id(a))
```

annab tulemuseks (näiteks):

```
1464424784
1464424816
1464424880
```

Näeme, et iga omistamisega salvestati muutujasse `a` uus viit, mis siis viitab ka uuele objektile.

Mis juhtub, kui esimese omistamise järel antakse sama viit ka teisele muutujale?

Enesetest:

Mis ilmub ekraanile?

```
a = 5
b = a
a = 7
print(b)
```

Vali 5

Vali 7

Vali Midagi muud

Vali Veateade

Ka viitade vaatamine kinnitab, et `a` ja `b` lähevad pärast muutujale `a` uue väärtuse andmist kumbki oma teed.

```
a = 5
b = a
print(id(a))
print(id(b))
a = 7
print(id(a))
print(id(b))
```

Seega võib öelda, et muutuja `a` sai uue väärtuse - vana väärtus oli 5 ja uus on 7. Samas seda väärtust 5 mälu täisarvulise objektina ei muudetud. Nüüd on muutujasse `a` salvestatud uus viit, mis viitab täisarvulisel objektile väärtusega 7. Päris hävingule see väärtusega 5 objekt ka mõeldud pole, sest muutujas `b` olev viit jääb talle viitama.

Samamoodi juhtus ka `a += 4` korral - uus väärtus uue viidaga.

Niisiis tegelikult pole täisarvu võimalik muuta. Kui seda tahaksime teha, siis tegelikult võetakse kohe uus objekt, mitte ei "parandata" vana. Sedalaadi andmetüüpe nimetatakse *mittemuteeritavateks*. Kui mingi objekt on loodud, siis selliseks ta jääb kogu oma eksisteerimise aja. Kui ta sellisena enam vajalik pole, siis visatakse ära. Eks nii kipub päriseluski tänapäeval mitmete asjadega olema.

Sõnu *muteeritav*, *mittemuuteritav* ja *muteerima* Õigekeelsussõnaraamatus pole. Samas on seal sõnad *muteeruma* ja *mutatsioon*. Seega kui me midagi muteerime, siis see asi muteerub ja toimub mutatsioon. Eks võiks ka
tavakeelsemalt *muutmisest*, *muutumisest* ja *muutusest* rääkida, aga programmeerimises kasutatakse täpsuse eesmärgil siiski veidi võõrapärasemaid termineid nagu ka Aivar Annamaa programmeerimise õpikus.

Mittemuteeritavad (ingl *immutable*) on näiteks arvud, sõned, ennikud ja tõeväärtused. Muteeritavad (ingl *mutable*) on aga näiteks järjendid, hulgad ja sõnastikud. Muteeritavate objektide puhul on olemas võimalused nende muutmiseks - muteerimiseks. Näiteks saab elemente lisada, eemaldada ja välja vahetada.

Eelmise alapunkti lõpus jäi õhku selgitus, et miks arvude, sõnede ja ennikute puhul, vähemalt mingitel juhtudel, antakse viit samale objektile ka siis, kui muutujatpidi (`b = a`) omistamist pole. Nt

```
c = 6
d = 6
```

Nimelt võidakse mittemuteeritavate tüüpide korral nii korraldada, et ükskõik millisele muutujale vastav väärtus antakse, siis see võetakse samast kohast mälus, seega viit on sama.

Kuna neid väärtusi nagunii muuta ei saa, siis pole muret sellega, et mingile väärtusele erinevate (ka omavahel seostamata) muutujate poolt viidatud on.

4.3 JÄRJENDI MUTEERIMINE

Pythonis on järjendid muteeritavad, mis tähendab, et järjend võib programmi töö jooksul sisaldada erinevaid väärtusi. Näiteks

```
järjend = [1, 2, 6, -7]
print(id(järjend))

järjend[1] = 4
print(id(järjend))

järjend.append(45)
print(id(järjend))

järjend += [9]
print(id(järjend))

järjend = järjend + [8]
print(id(järjend))
```

Väljastatakse arvud

```
20139104
20139104
20139104
20139104
9977904
```

Näeme, et kuni viimase muutuseni on tegemist sama järjendiobjekti muutmisega, viimane aga tekitab uue.

Mis juhtub, kui enne muteerimist on ka teine muutuja seotud?

Enesetest:

Mis ilmub ekraanile?

```
a = [1, 3, 5, 6]
b = a
a[1] = 7
print(b[1])
```

Vali 3

Vali 7

Vali Midagi muud

Vali Veateade

Enesetest:

Mis ilmub ekraanile?

```
a = [1, 3, 5, 6]
b = a
b[1] = 7
print(a[1])
```

Vali 3

Vali 7

Vali Midagi muud

Vali Veateade

Näeme, et tõesti on tegemist täpselt sama objektiga. Vahel on aga vaja, et koopiad oleksid sõltumatud.

Sõltumatu koopia

Kui tahame sõltumatut koopiat, siis saame kasutada viilutamist

```
c = a[:]
```

või funktsiooni `copy`

```
d = a.copy()
```

Enesetest:

Mis ilmub ekraanile?

```
a = [1, 3, 5, 6]
b = a[:]
b[1] = 7
print(a[1])
```

Vali 3

Vali 7

Vali Midagi muud

Vali Veateade

See sõltumatus on siiski vaid välimisel tasemel ja toimib kenasti, kui järjend koosneb mittemuteeritavatest suurustest. Kui aga järjend koosneb muteeritavatest elementidest, näiteks järjenditest, siis sügavamal tasemel jääb sõltuvus alles. See tuleb sellest, et viidad välimistele elementidele on samad.

```
tabel = [[1, 2, 3], [4, 5, 6]]
tabel2 = tabel.copy() # või tabel2 = tabel[:]
print(id(tabel[0]))
print(id(tabel2[0]))
```

Väljastatakse

```
18374408
18374408
```

Kui nüüd muudame näiteks ühel kahemõõtmelisel järjendil ülal vasakul asetsevat elementi, siis muutub see ka teisel.

```
tabel = [[1, 2, 3], [4, 5, 6]]
tabel2 = tabel.copy() # või tabel2 = tabel[:]
# print(id(tabel[0]))
# print(id(tabel2[0]))
tabel2[0][0] = 12
print(tabel)
print(tabel2)
```

Väljastatakse

```
[[12, 2, 3], [4, 5, 6]]
[[12, 2, 3], [4, 5, 6]]
```

Samas välimisel tasemel on sõltumatus ja nii saame elementjärjendeid eemaldada, uusi elementjärjendeid juurde panna. Saame ka elementjärjendeid uutega asendada.

```
tabel = [[1, 2, 3], [4, 5, 6]]
tabel2 = tabel.copy() # või tabel2 = tabel[:]
tabel[1] = [23, 45, 67]
print(id(tabel[1]))
print(id(tabel2[1]))
print(tabel)
print(tabel2)
```

Väljastatakse

```
11943984
12598368
[[1, 2, 3], [23, 45, 67]]
[[1, 2, 3], [4, 5, 6]]
```

Kuna nüüd on kummaski järjendis indeksiga 1 elementjärjent erinev (erineva viidaga), siis saame ka selles elementjärjendis ka üksikuid elemente sõltumatult muuta.

```
tabel = [[1, 2, 3], [4, 5, 6]]
tabel2 = tabel.copy() # või tabel2 = tabel[:]
tabel[1] = [23, 45, 67]
tabel[1][1] = -4
print(tabel)
print(tabel2)
```

Kui tahta täielikult sõltumatut koopiat, siis saab kasutada funktsiooni `deep.copy` moodulist `copy`.

```
from copy import deepcopy
tabel3 = [[1, 2, 3], [4, 5, 6]]
tabel4 = deepcopy(tabel3)
tabel3[0].append(7)
print(tabel3)
print(tabel4)
```

Järjendite "korrutamine"

Pythonis on mitmeid võimalusi, mida teistes programmeerimiskeeltes ei pruugi olla. Näiteks tehtemärk `*` toimib ka sõnede, järjendite ja ennikute korral.

```
print([1, 3, 4] * 3)
print("Elagu! " * 4)
print((1, 4, 6) * 2)
```

Kordaja võib olla ka ees, nt `4 * "Elagu"`.

Proovime nüüd *-märgi abil saada 3*4-maatriksi, mis sisaldab ainult nulle. Kuna `[0] * 4` annab meile järjendi, milles on 4 nulli, siis `[[0] * 4] * 3` toimel soovitu saamegi.

```
maatriks = [[0] * 4] * 3
print(maatriks)
```

Väljastatakse maatriks

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Selline maatriks võib olla heaks aluseks mingite andmete jaoks, mida siis järjest õigetele kohtadele pannakse.

Proovime näiteks kohale `(0, 0)` panna arvu `1`.

```
maatriks = [[0] * 4] * 3
maatriks[0][0] = 1
print(maatriks)
```

Väljastatakse maatriks

```
[[1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0]]
```

Näeme, et muutus toimus kõikides ridades. Seda sellepärast, et *-märgi toimel kopeeritakse vastav arv korda viitasid. Kuna need viitavad samale järjendile, siis muidugi muutuvad kõik korraga.

Kui tahta ikkagi sõltumatuid ridu saada, siis võime kirjutada järgmise funktsiooni.

```
def nullmaatriks(ridu, veerge):
    maatriks = []
    for i in range(ridu):
          maatriks.append([0] * veerge)
    return maatriks

maatr = nullmaatriks(3, 4)
maatr[0][0] = 1
print(maatr)
```

4.4 FUNKTSIOON JA MUTEERIMINE

Järjenditega tegutsemiseks on erinevad funktsioone ja muidugi saame neid juurde luua. Funktsioon ja järjend saavad olla seotud mitmel moel. Enamasti on Pythonis juba olemasolevad järjendite funktsioonid seotavad järjendiga punkti abil, nt

```
jarjend.append()
jarjend.insert()
jarjend.count()
jarjend.copy()
jarjend.reverse()
jarjend.sort()
jarjend.clear()
```

Selliseid funktsioone nimetatakse meetoditeks. Tegemist on objektorienteeritud lähenemisega, millest on juttu ka selle nädala silmaringimaterjalis.

Meie aga loome funktsioone, kus järjend antakse ette argumendina. Ükskõik kuidas järjendiga seos korraldatud on, tuleb silmas pidada, kas see järjend ise funktsiooni töö jooksul muutub või mitte.

Koostame kaks funktsiooni, mis nagu teeksid sama asja, aga ühel juhul argumendina antud järjend muutub ja teisel mitte.

Olgu ülesandeks argumendina antud järjendi elementide nullimine. Esialgu teeme sellise funktsiooni, mis muudab argumendina antud järjendit.

```
def muuda_nullideks(jarj):
    for i in range(len(jarj)):
        jarj[i] = 0

jarjend = [2, 5, -2, 4]
muuda_nullideks(jarjend)
print(jarjend)
```

Kui vaatame, mis väärtuse see funktsioon tagastab, ja mis tüüpi see väärtus on, siis saame:

```
print(muuda_nullideks(jarjend))
print(type(muuda_nullideks(jarjend)))
```

mis annab tulemuse

```
None
<class 'NoneType'>
```

Niisiis ei tagastata funktsioon nagu midagi - `None` just seda märgibki. Kui nüüd tahaksime selle funktsiooni väärtuse uuele muutujale anda, siis saab sellegi väärtuseks `None`. Sellise funktsiooni puhul ongi olulisem see n-ö kõrvalefekt, milleks on antud juhul järjendi muutmine.

Teeme nüüd funktsiooni, mis tagastaks vastava nullitud järjendi ja argumendina antud järjendit ei muudaks:

```
def tagasta_nulliline(järj):
    uus_järj = []
    for i in range(len(järj)):
        uus_järj.append(0)
    return uus_järj


järjend2 = [4, -5, 2, 3]
print(tagasta_nulliline(järjend2))
print(järjend2)
```

Nüüd saame ka funktsiooni rakenduse väärtuse uuele muutujale anda:

```
uus_järjend = tagasta_nulliline(järjend2)
```

Saab koostada ka funktsiooni, mis nii muudab argumenti, kui ka tagastab väärtuse. See võib aga olla murede allikaks, sest argumenti muutev kõrvalefekt võib olla häiriv ja üllatuslik.

```
def muuda_nullideks_tagasta(järj):
    for i in range(len(järj)):
        järj[i] = 0
    return järj

järjend3 = [7, -3, -1, 6]
print(muuda_nullideks_tagasta(järjend3))
print(järjend3)
```

Laiendame neid programme kahemõõtmelise järjendi juhule.

```
def muuda_nullideks2D(jär2d):
    for i in range(len(jär2d)):
        for j in range(len(jär2d[i])):
            jär2d[i][j] = 0

tabel = [[1, 5, 6], [-5, 7, 2]]
muuda_nullideks2D(tabel)
print(tabel)

def tagasta_nulliline2D(jär2d):
    uus_jär2d = []
    for i in range(len(jär2d)):
        uus_rida = []
        for j in range(len(jär2d[i])):
            uus_rida.append(0)
        uus_jär2d.append(uus_rida)
    return uus_jär2d


tabel2 = [[-1, 2, 3], [5, -1, 3]]
print(tagasta_nulliline2D(tabel2))
print(tabel2)
```

4.5 KORDAMINE

Töövoost, mida ikka kasutame

Vaatleme nüüd kordavalt töövoogu, mis mõningate variatsioonidega võib ette tulla paljude ülesannete puhul.

Olgu algandmed teatud kujul failis. Näiteks on vaadeldavate objektide kohta andmed ridade kaupa esitatud. Ridadel on eraldajateks tühikud või mingid muud sümbolid, näiteks semikoolonid. Selleks et üldse andmetega toimetada, peab programm need sisse lugema. Sisselugemisel võib juba midagi nende andmetega ette võtta, näiteks kuidagi teisendada või mõned andmed üldse välja jätta.

Sisselugemisel peaks andmed paigutama sobivasse andmestruktuuri näiteks kahemõõtmelisse järjendisse või sõnastikku.

Nüüd saab selle struktuuri abil juba andmeid analüüsida. Vajadusel saab näiteks nende andmete põhjal ka mingi teise andmestuktuuri tekitada. Ja siis ka juba tollega midagi teha.

Lõpuks saab tulemused ekraanile väljastada või hoopis mingitesse failidesse kirjutada.

Püüame nüüd näite varal kõik need etapid läbi teha.

Olgu andmed mingile üritusele, näiteks Tartusse laulupeole tulijate kohta. Olgu igal real kirjas maakond, selle keskuse kaugus Tartust, mitu meest on tulemas ja mitu naist.

```
Tartumaa : 0 : 302 : 400
Hiiumaa : 293 : 24 : 44
Harjumaa : 186 : 350 : 300
Ida-Virumaa : 131 : 280 : 270
Lääne-Virumaa : 123 : 200 : 160
Raplamaa  : 149 : 56 : 46
Põlvamaa : 49 : 87 : 65
Viljandimaa : 78 : 170 : 170
Saaremaa : 322 : 154 : 128
Läänemaa : 238 : 105 : 102
Jõgevamaa : 53 : 79 : 82
Võrumaa : 70 : 75 : 89
Valgamaa : 90 : 65 : 76
Järvamaa : 103 : 124 : 120
Pärnumaa : 170 : 143 : 145
```

Küsime faili nime.

Failist sisselugemisel tahame andmed panna sõnastikku, kus võtmeks on maakonna nimi ja väärtuseks ennik, mis koosneb ülejäänud andmetest, mis maakonna kohta teada on. Seejuures paneme ennikusse kahekordse kauguse (Tartusse ja tagasi).

Teeme selleks funktsiooni, mis saab argumendiks failinime ja tagastab nõutud sõnastiku.

Arvutame, mitu kilomeetrit sõidavad kõik inimesed kokku.

Seejärel teeme selle sõnastiku põhjal kahemõõtmelise järjendi, mille ühes reas on meeste arvud ja teises reas naiste arvud. Selle järjendi põhjal arvutame, mitmes maakonnas on tulijate hulgas mehi rohkem, kui naisi.

Küsime faili nime, kuhu kirjutada.

```
def failist_sonastikku(failinimi):
    fail = open(failinimi)
    maakondade_sonastik = {}

    for rida in fail:
        #Teeme koolonite juurest katki
        jupid = rida.split(":")

        #Leiame väärtused järjendist
        maakond = jupid[0].strip()
        kaugus = int(jupid[1]) * 2 #edasi ja tagasi
        mehed = int(jupid[2])
        naised = int(jupid[3])

        #Moodustame sõnastiku
        ennik = (kaugus, mehed, naised)
        maakondade_sonastik[maakond] = ennik

    fail.close()

    return maakondade_sonastik

def sonastikust_kauguste_summa(maakondade_sonastik):
    summa = 0

    for maakond, vaartused in maakondade_sonastik.items():
        kaugus = vaartused[0]
        mehi = vaartused[1]
        naisi = vaartused[2]

        summa += kaugus * mehi + kaugus * naisi

    return summa
```

```
def sonastikust_tulijate_jarjend(maakondade_sonastik):
    mehed = []
    naised = []

    for maakond, vaartused in maakondade_sonastik.items():
        mehed.append(vaartused[1])
        naised.append(vaartused[2])

    return [mehed, naised]

def mehi_rohkem_kui_naisi(maakondade_sonastik):
    tulijad = sonastikust_tulijate_jarjend(maakondade_sonastik)
    maakondade_arv_kus_mehi_rohkem = 0

    for i in range(len(tulijad[0])): #teame, et mõlemad järjendid on
sama pikad
        mehi = tulijad[0][i]
        naisi = tulijad[1][i]
        if mehi > naisi:
            maakondade_arv_kus_mehi_rohkem += 1

    return  maakondade_arv_kus_mehi_rohkem
```

Sama programm hoopis funktsiooniga

Konkreetset probleemi saab lahendada mitmel erineval moel. Nüüd vaatlemegi sellist tüüpi ülesannet, kus on programm antud ja on ka öeldud, mida see programm teeb. Antud on ka teine programm, mis peaks tegema sama, aga kasutab seejuures ühte funktsiooni, millest on aga teada vaid nimi. Ülesandeks ongi nüüd koostada nõutud funktsioon, et kaks programmi võrdväärselt töötaksid.

Proovime siin ühte sellis ülesannet lahendada.

Järgnev programmilõik leiab kahemõõtmelise järjendi korral, kui paljudes ridades on positiivseid elemente.

```
positiivsega_ridu = 0
for i in range(len(a)):
    leidub_positiivne = False
    for j in range(len(a[i])):
        if a[i][j] > 0:
            leidub_positiivne = True
            break
    if leidub_positiivne:
        positiivsega_ridu += 1
print(positiivsega_ridu)
```

Koostada funktsioon on_positiivseid, mille puhul alltoodud programmilõik töötaks ülaltooduga võrdväärselt.

```
positiivsega_ridu = 0
for i in range(len(a)):
    if on_positiivseid(a[i]):
        positiivsega_ridu += 1
print(positiivsega_ridu)
```

On selge, et selleks, et esimene programm üldse töötaks peab enne antud lõiku olema defineeritud kahemõõtmeline arvude järjend `a`.

Võiksime võtta testimiseks sellise järjendi, mille osades ridades on positiivseid elemente, aga osades mitte.

Ennekõike tuleb `i` põhjalikult selgeks saada, mis toimub esimeses programmis, siis on lootust teine programm samamoodi tööle saada. Paneme ka tähele, et mõlema programmi algus ja lõpp on täiesti samad.

Kui ridarealt läbi analüüsida, saame järgmised ideed.

```
positiivsega_ridu = 0

# Paneme tähele, et see suurus lõpuks väljastatakse

for i in range(len(a)):

# len(a) on 2d järjendi pikkus ehk ridade arv
# Seega välimine tsükkel võtab igal sammul uue reaindeksi

    leidub_positiivne = False

# Igal välimise tsükli sammul saab leidub_positiivne väärtuseks
False

    for j in range(len(a[i])):

# len(a[i]) on konkreetse rea elementide arv
# Sisemine tsükkel vaatab konkreetse rea elemente järjest

        if a[i][j] > 0:
            leidub_positiivne = True
            break

# Kui vaadeldav element on positiivne, siis saab leidub_positiivne
# väärtuseks True ja sisemisest tsüklist tullakse välja, sest ongi
# positiivne leitud ja edasi pole vaja otsida

    if leidub_positiivne:
        positiivsega_ridu += 1

# Kui konkreetse reaga on klaar, siis otsustatakse, kas tuleb
```

```
# suurendada loendajat

print(positiivsega_ridu)

#  Lõpuks loendaja väärtus väljastatakse.
```

Kui vaatame nüüd teist programmi, siis näeme, et selle kontroll, kas konkreetses reas on positiivseid elemente, on usaldatud funktsioonile `on_positiivseid`.

```
positiivsega_ridu = 0
for i in range(len(a)):
    if on_positiivseid(a[i]):
        positiivsega_ridu += 1
print(positiivsega_ridu)
```

Mõtleme läbi, mis tüüpi on funktsiooni `on_positiivseid` argument ja mis tüüpi väärtus tuleb tagastada. Näeme, et funktsioonile antakse ette `a[i]`, mis antud kontekstis on kahemõõtmelise järjendi rida ehk siis arvude järjend. Paneme tähele, et seda funktsiooni kasutatakse *if*-lause tingimusena. Seega peab tagastatav väärtus olema tõeväärtustüüpi - `True` või `False`.

`True` peaks tagastatama, kui selles reas on vähemalt üks positiivne element ja `False` vastasel juhul. Lahenduseks sobib näites selline variant.

```
def on_positiivseid(rida):
     for i in range(len(rida)):
         if rida[i] > 0:
                return True
     return False
```

Palun katsetage ja veenduge selles ise.

4.5 OBJEKT-ORIENTEERITUD PROGRAMMEERIMINE

Klassid kui tabelid

Eelmisel nädalal vaatasime, kuidas andmeid hoitakse andmebaasides ning kuidas neid andmeid andmebaasist kätte saada. Sellel nädalal vaatame, kuidas andmeid saab hoida objektides.

Objektis informatsiooni hoidmiseks peame läbi mõtlema, kuidas soovime objektis andmeid kirjeldada. Sarnaste objektide jaoks ühiste reeglite sätestamiseks kasutame klasse. Klassid on nagu andmebaaside tabelid, mis ütlevad, millist infot me iga tabelis oleva isendi kohta teame. Klassi defineerimiseks kasutatakse märksõna `class`.

Meenutame `Õppija` tabelit: igal `Õppijal` on enda `matriklinumber`, `eesnimi`, `perekonnanimi` ja `isikukood`. Need neli tunnust võime lugeda muutujateks, mida väärtustame iga `Õppija` jaoks eraldi.

Muutujaid, mis kehtivad iga isendi enda kohta, nimetatakse isendimuutujateks ehk isendiväljadeks. Pythonis kasutatakse nende klassisiseseks eristamiseks märksõna `self`.

Nõnda saame klassis `Õppija` defineerida muutujad `self.matrikli_nr`, `self.eesnimi`, `self.perenimi` ja `self.isikukood`. Klassist väljaspool nende muutujate kasutamisel kasutatakse `self` märksõna asemel vastavat isendit hoidva muutuja nime. `self` märksõna kasutatakse ka isendi informatsiooni kasutavate klassis defineeritud funktsioonide esimese argumendina. Ühte levinud isendispetsiifilist funktsiooni vaatleme järgmisena.

Uue isendi loomine

Uue isendi loomisel soovime need isendimuutujad väärtustada ehk omistada neile mingid väärtused. Selleks kasutatakse konstruktorit, mis on klassile iseloomulik funktsioon. Konstruktorit kasutatakse iga isendi loomisel isendimuutujate väärtustamiseks. Funktsiooni nimeks on `__init__`, mis tuleneb uue isendi initsialiseerimisest ehk lähtestamisest. Funktsiooni parameetriteks on esmalt märksõna `self` ning seejärel kõik teised muutujad, mida soovime isendisse salvestada.

```
class Õppija:
    def __init__(self, mat_nr, eesnimi, perenimi, isikukood):
        self.matrikli_nr = mat_nr
        self.eesnimi = eesnimi
        self.perenimi = perenimi
        self.isikukood = isikukood
```

Märkame, et `self.eesnimi` muutuja on isendispetsiifiline, kuid eesnimi muutuja pärineb funktsiooni parameetritest. Kui kasutada sama nimega muutujaid, tuleb olla eriti tähelepanelik, millist muutujat me igal hetkel kasutame. Kui kogemata muutujad segamini ajame, võib meie loodav programm käituda ettearvamatult. Loodud konstruktorit saame kasutada uue isendi loomisel. Kui konstruktori `__init__` defineerimine oli nagu andmebaaside `CREATE TABLE` käsk, siis konstruktori kasutamine on analoogne `INSERT INTO` käsuga:

```
õppur1 = Õppija("A034", "Albert", "Paas", 34105212737)
õppur2 = Õppija("A037", "Pearu", "Murakas", 34206122154)
print(õppur1.eesnimi)
print(õppur2.eesnimi)
```

Sarnaselt üksikute muutujate lugemisele võime isendimuutujaid ka ümber väärtustada või lisada.

```
õppur1.lemmikloom = "koer"
print(õppur1.lemmikloom)
```

Väljastatakse `"koer"`.

Isendispetsiifilised funktsioonid

Kui soovime välja printida informatsiooni isendi kohta, siis üheks variandiks on üksikute isendiväljade kasutamine. `print(õppur1)` annab aga hetkel eriskummalise tulemuse `<__main__.Õppija object at 0x02BFBE90>` , mis ütleb, et tegu on isendiga klassist `Õppija` ning see asub mäluväljal `0x02BFBE90`. Selleks, et `print` käsk annaks meile informatiivsema kirjelduse, tuleb klassis defineerida isendispetsiifiline funktsioon `__str__`:

```
def __str__(self):
    return "Õppija nimi: "+self.eesnimi+" "+self.perenimi+"."
```

Kui soovime, võime isendispetsiifilisi funktsioone juurde teha. Sellisel juhul võime sama funktsiooni välja kutsuda mistahes `Õppija` isendi peal, saades isendi andmetest sõltuva tulemuse.

```
def sugu(self):
    if self.isikukood//10000000000%2 == 1: #Vaatleme, kas isikukoodi 
esimene number on paaris- või paaritu arv.
        return "M"
    else:
        return "N"
```

Funktsioonide väljakutsumine on sarnane sõnede peal `split` ja `strip` funktsioonide kasutusega:

```
print(õppur1.sugu())
```

Staatilised muutujad ja funktsioonid

Lisaks isendispetsiifilistele muutujatele ja funktsioonidele võib klassis defineerida ka kõikidele sama klassi isenditele ühiseid muutujaid ja funktsioone. Neid nimetatakse staatilisteks, kuna nad ei muutu sõltuvalt isendist. Staatilisi muutujaid võib võrrelda globaalsete muutujatega, mispuhul isendispetsiifilised muutujad on sarnased lokaalsete muutujatega - kasutatakse funktsioonides.

Sarnaselt globaalsete muutujatega deklareeritakse staatilised muutujad klassi sees funktsiooniväliselt. Erinevalt isendispetsiifiliste muutujatest tuleb staatiliste muutujate lugemiseks või muutmiseks pöörduda isendi asemel terve klassi poole. Kui meie näites oleks eesnimi staatiline muutuja, tuleks `õppur1.eesnimi` asemel kasutada `Õppija.eesnimi`. Seejuures saab, kuid ei ole soovitatav, kasutada sama nimega staatilisi ja isendispetsiifilisi muutujaid.

Staatilised funktsioonid erinevad isendispetsiifilistest vaid paari omaduse poolest: parameetrites ei kasutata märksõna `self`, funktsioonis sees ei saa mittestaatilisi isendimuutujaid kasutada ning funktsiooni väljakutsumiseks kasutatakse isendi muutujanime asemel klassi üldnimetust. Võime lisada klassi algusesse ühe muutuja `kool`, ühe staatilise funktsiooni ning muuta väljaprinditavat infot:

```
class Õppija:
    kool = "Tartu Ülikool"
    def kusÕpib():
        return "Õpib koolis "+Õppija.kool+"."
    def __str__(self):
        return "Õppija nimi: "+self.eesnimi+"
"+self.perenimi+".\n"+Õppija.kusÕpib()
Nüüd on huvitav uurida print(õppur1) ja print(õppur2) väljundit.
```

Lõpetuseks

Oleme vaadelnud, kuidas ehitada objekt erinevate omadustega. Objekte on hea kasutada olukordades, kus ühe ja sama väärtusega on seotud hästi palju teisi. `Õppija` klassi asemel oleksime saanud kasutada ka näiteks kolme sõnastikku, kus ühes on matriklinumbrile vastav eesnimi, teises perekonnanimi, kolmandas isikukood. Ühendades kõik ühte isendisse muutub kood palju kergemini loetavaks ja hallatavamaks. Tähele võib panna ka võimalust panna isendeid teistesse andmestruktuuridesse nagu järjend, mispuhul ei ole vaja iga uue isendi jaoks uut muutujanime, piisab järjendi nimest ja indeksist. Sellisel juhul saab isendeid tekitada tsüklitega.

Näidiskoodi lõppkuju:

```
class Õppija:
    kool = "Tartu Ülikool"
    def kusÕpib():
        return "Õpib koolis "+Õppija.kool+"."
    def __init__(self, mat_nr, eesnimi, perenimi, isikukood):
        self.matrikli_nr = mat_nr
        self.eesnimi = eesnimi
        self.perenimi = perenimi
        self.isikukood = isikukood
    def __str__(self):
        return "Õppija nimi: "+self.eesnimi+"
"+self.perenimi+".\n"+Õppija.kusÕpib()
    def arvuta(a,b):
        return a+b
    def sugu(self):
        if self.isikukood//10000000000%2 == 1:
            return "M"
        else:
            return "N"
#Konstruktori kasutus
õppur1 = Õppija("A034", "Albert", "Paas", 34105212737)
õppur2 = Õppija("A037", "Pearu", "Murakas", 34206122154)

#Isendimuutuja tekitamine/muutmine
õppur1.lemmikloom = "koer"
print(õppur1.lemmikloom)

#Isendispetsiifiline funktsioon
print(õppur1.sugu())

#Väljastamine, kasutades isendispetsiifilist funktsiooni koos
staatilise muutuja ja funktsiooniga
print(õppur1)
print(õppur2)

#Muu staatilise funktsiooni kasutus
print(Õppija.arvuta(1,2))
```