Sisukord

Kursuse "Teeme ise arvutimänge - algus"

3. RAAMAT

TSÜKLID TSÜKLITES JA LISTID

Tiina Kull

Tartu Ülikool

2012

Ussimäng

Kindlasti juba ootad, millal jõuame graafiliste mängude loomise juurde. Kohe saad sellega algust teha, kuigi päris graafika õppimiseni jõuame alles 5. nädalal.

Mängude loomise õppimine käib meil ikka läbi praktiliste ülesannete, mis sageli tähendab täiesti võõra koodi ümber kirjutamist ja selles orienteeruda püüdmist. Nii ka nüüd. Kirjuta alloleva u 30 reaga ussimängu esmaversioon ümber oma Idle tekstiredaktorisse, kindlasti salvesta uus fail kohe näiteks pealkirjaga ***uss1.py***. Ära unusta salvestamast ka trükkimise käigus, jube tüütu oleks ju koodi tippimist teha mitu korda.

Jaaaa... ma natuke kujutan ette sinu ära vajunud nägu ja küsimust, mida!, miks peab jälle ümber kirjutama...? Usu, see on üks paremaid viise, kuidas sa tegelikult ka mõtled koodi struktuuri peale märksa süvenenumalt kui lihtsalt kopeerides ja hakkad nagu iseenesest juurdlema iga rea korral, miks seda just nii on kirjutatud. Nii et see on ainult kasulik.

Kui oled koodi trükkimisega hakkama saanud, siis pane programm kindlasti käima. See veel väga palju suurt midagi ei tee, kuid alustuseks juba midagi küll. Proovi tõlgendada faili kirjutatut, mida iga rida selles koodis võiks tähendda. Mõned asjad on uued, kuid põhikonstruktsioonid peaksid juba tuttavad olema. Palju on kasutatud graafika mooduli ***pygame*** käske, nendest räägime täpsemalt u 5. nädalal. Edaspidi hakkame sellele koodile programmilõike juurde lisama ja muutma, et lõpuks päris mäng kokku tuleks.

Ahaa, veel enne kui koodi käima paned, tuleb sul üks pilt ***pea2.png*** enne samasse kataloogi salvestada, kuhu sa salvestad ka *uss1.py* koodi. Pilt on see sama rohelise peaga ja krõllis silmadega Pythonike järgmisel real.

Põhimõtteliselt võid ka nii teha, et kui sulle see konkreetne pilt ei meeldi, siis otsi endale meelepärasem ja salvesta täpselt sama nimega nagu originaalpilt samasse kataloogi. Kui sa seda ei tee, siis lihtsalt hakkab programm veateateid pilduma.

Kõige enne aga tuleb Pythoni programmile juurde installeerida ***pygame*** moodul. See on väikeste abiprogrammide kogumik, mis aitab palju lihtsamalt (mitte nii palju ise jalgratast leiutades) graafilisi mänge programmeerida.

Juhend, *pygame* installeerimiseks:

- Mine Pygame'i kodulehele: http://www.pygame.org/news.html
- Vali ülevalt vasakust nurgas *Downloads*
- Vastavalt oma arvuti operatsioonisüsteemile ja *Pythoni* versioonile vali pygame versioon ja vajuta vastaval lingil. Pygame ja Python peavad olema sama bittide arvuga, kusjuures, kui oled eelnevalt 64bitise alla laadinud, siis tuleks see muuta 32 peale, sest pygame on max 32 peale tehtud (veel).
- Installeeri *Pygame* samasse kataloogi, kus asub sul ka *Python.*

NB! Üks ebamugav asi seondub aga pygame-ga. Nimelt ei saa IDLE ja pygame omavahel väga hästi läbi. Käima saab panna ja programme saab samuti kirjutada, kuid pygame käivitumisel jääb pygame tihti kinni. See tähendab aga lihtsalt ühte väikest lisaliigutust - sulgeda iga kord peale mängimist pygame. Alati tasub ka koodi lõppu panna käsk quit(), mis osades olukordades leevendab pygame kinni jooksmist.

Lahendus on kasutada ka cmd.exe-t oma programmi käima panemisel ja mitte Idle-t, kuid see pole ka just eriti mugav. Kes tahab, võib teistele cmd käske foorumis õpetada.

Ja siin esimene osa ussimängu koodist:

```
#!/usr/bin/env python

from pygame import *
from sys import *

init()
screen = display.set_mode([640,480])
pea_pilt = image.load('pea2.png') # 85x96 pics
samm = 10
x = 50
y = 50
while True:
        screen.fill([255,255,255])
        screen.blit(pea_pilt,[x,y])
        display.flip()
        time.delay(10)
        for e in event.get():
                if e.type == QUIT:
                        exit()
                elif e.type == KEYDOWN:
                        if e.key == K_UP:
                                y = y - samm
                        elif e.key == K_DOWN:
                                y = y + samm
                        elif e.key == K_LEFT:
                                x = x - samm
                        elif e.key == K_RIGHT:
                                x = x + samm
```

vol2

Loodan, et kui graafika käsud välja jätta, siis said ussimängu koodist enam-vähem aru ja vähemalt katse-eksituse meetodil said ussipea ekraanil liikuma.

Kui sa oled piisavalt saanud katsetada ussi liikumist, siis lisa esialgsele koodile veel üks koodilõik. Võrdle koodi **uss1** nüüd järgmise koodiga **uss2** ja kirjuta puuduvad osad enda koodi juurde. Uuri, mis nüüd teisiti on ja miks uss nüüd just nii käitub?

```
#!/usr/bin/env python

from pygame import *
from sys import *

VASAKULE = 1
YLES = 2
PAREMALE = 3
ALLA = 4
init()
screen = display.set_mode([640,480])
pea_pilt = image.load('pea2.png') # 85x96 pics
samm = 10
x = 50
y = 50
liigub = PAREMALE
while True:
        screen.fill([255,255,255])
        screen.blit(pea_pilt,[x,y])
        display.flip()
        time.delay(10)
        if liigub == YLES:
                y = y - samm
        elif liigub == ALLA:
                y = y + samm
        elif liigub == VASAKULE:
                x = x - samm
        elif liigub == PAREMALE:
                x = x + samm
        if x<0:
                x = 0
        if x>600:
                x = 600
        if y<0:
                y = 0
        if y>430:
                y = 430
        for e in event.get():
                if e.type == QUIT:
                        exit()
```

```
                elif e.type == KEYDOWN:
                        if e.key == K_UP:
                                liigub = YLES
                        elif e.key == K_DOWN:
                                liigub = ALLA
                        elif e.key == K_LEFT:
                                liigub = VASAKULE
                        elif e.key == K_RIGHT:
                                liigub = PAREMALE
```

vol3

Ja veel üks uus iseseisev graafikaõppimine sellel nädalal. Huvitav, mida järgmine kood teeb?

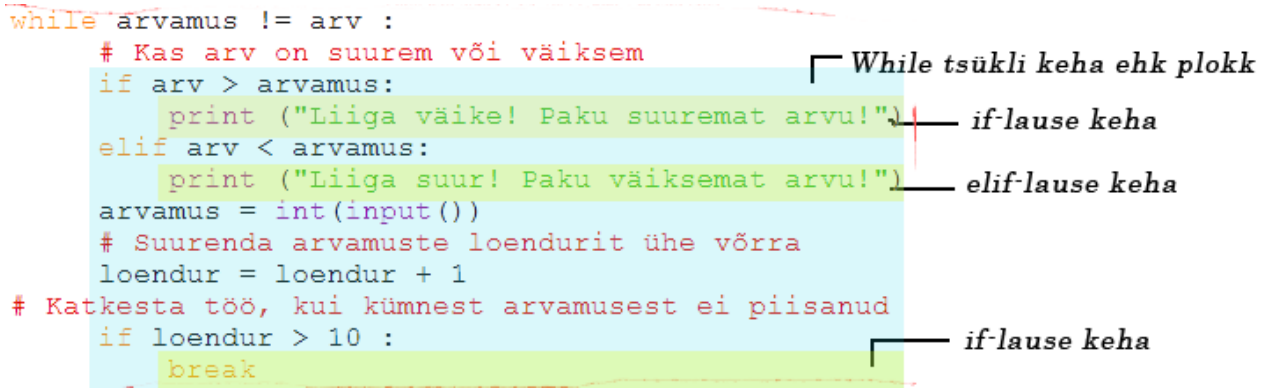
Lae uss3-ga samasse kausta ka pilt bang2.png 


```
#!/usr/bin/env python

from pygame import *
from sys import *

VASAKULE = 1
YLES = 2
PAREMALE = 3
ALLA = 4
init()
screen = display.set_mode([640,480])
pea_pilt = image.load('pea2.png')
bang_pilt = image.load('bang2.png')
samm = 10
x = 50
y = 50
liigub = PAREMALE
bang = False
while True:
        screen.fill([255,255,255])
        if bang:
                screen.blit(bang_pilt,[x,y])
        else:
                screen.blit(pea_pilt,[x,y])
        display.flip()
        time.delay(10)
        if not bang:
                if liigub == YLES:
                        y = y - samm
                elif liigub == ALLA:
                        y = y + samm
                elif liigub == VASAKULE:
                        x = x - samm
                elif liigub == PAREMALE:
                        x = x + samm
        if x<0:
                x = 0
                bang = True
        if x>600:
                x = 600
                bang = True
        if y<0:
                y = 0
                bang = True
        if y>430:
                y = 430
                bang = True
```

```
        for e in event.get():
                if e.type == QUIT:
                        exit()
                elif e.type == KEYDOWN:
                        if e.key == K_UP:
                                liigub = YLES
                        elif e.key == K_DOWN:
                                liigub = ALLA
                        elif e.key == K_LEFT:
                                liigub = VASAKULE
                        elif e.key == K_RIGHT:
                                liigub = PAREMALE
```

Topelt tsükkel

Kordame kõigepealt veelkord üle, mis asi oli tsükli keha või if-lause keha ehk käskude plokk. Heaks näiteks on järgnev joonis, kus on mitmed plokid üksteise sees.



Täpselt sama moodi nagu on üleval näites tsükli sisse pandud teisi if-konstruktsioonide osi võib ühe tsükli sisse panna ka teisi tsükleid. Miks see hea peaks olema? Toon sulle ühe näite sissejuhatuseks ja siis proovime leida analoogiaid mängude maailmast.

Tsüklite õppimisel tõin sulle korrutustabeli tegemise näite.

```
for i in range(5):
    print(i, "x 7 on", i*7)
```

See koodijupp kirjutas välja aga vaid seitsmega korrutised ja sedagi kuni 5-ni. Mis korrutustabel see selline on? Korrutustabelis on ju toodud arvu korrutised palju rohkemate arvudega?

Asja parandamiseks tulebki võtta kasutusele teine tsükkel esimese tsükli ümber. Teine tsükkel hakkab ühte ja seda sama korrutiste seerjat välja kirjutama, kuid iga kord uue arvuga.

Proovime! Ava oma Idle tekstiredaktor ja kirjuta, mõtle kaasa ning pane käima:


```
for tegur in range(2, 5): # teen esialgu korrutustabeli 2, 3 ja 4-ga
    for i in range(1, 11):
        print(i,"x", tegur, "=", i*tegur)
    print() # see trükib tühja rea, kui sisemine tsükkel on läbi
```

Kuidas topelttsükkel töötab?

- Kõigepealt läheb käima välimine tsükkel, mille esimene väärtus on 2.
- Selle ühe väärtusega (2) täidetakse kogu ülejäänud tsükli sisu ehk sisemine tsükkel + tühi rida.
- Enne tühja reani ei jõuta, kui ka sisemine tsükkel on kõik oma tiirud ära teinud. Sisemine tsükkel teeb 10 tiiru (NB! `range(1-11) teeb 10 tiiru!`). Igal tiirul kirjutab ta välja ühe korrutustabeli rea kahega korrutamiseks. **`tegur = 2`** ja **`i`** saab igal sisemise tsükli tiirul järjest väärtused 1-st 10-ni.
- Kui sisemine tsükkel on lõpetanud, jõuab järg kätte tühja **`print()`** käsuni - lisatakse tabelite vahele 1 tühi rida.
- Kuna välimine for-tsükkel ei ole kõiki oma väärtuseid **`range()`** listist läbi käinud, siis minnakse nüüd uuele (järgmisele) välimise tsükli tiirule, kus muutuja **tegur** saab uue väärtuse 3, **`tegur=3`**.
- Nüüd korratakse uuesti sisemist tsüklit, kuid teguri väärtusega 3. Täpselt sama moodi jälle 10 sisemist tiiru. Kui sisemine tsükkel läbitud, lisatakse uus tühi rida.
- Viimane väärtus välimises tsüklis on 4. Ka neljaga tehakse läbi samad protseduurid.
- Nüüd enam välimises tsüklis elemente ei ole ja programm lõpetab töö.

Näited

Samasuguse ülesehitusega nagu oli korrutustabeli koostamine, võime kinnistavaks näiteks tuua ka tärnide trükkimise:

```
print("Mitu tärni kirjutan ühte ritta?")
tarnide_arv = int(input())
print("Mitu rida trükin?")
ridade_arv = int(input())

for rida in range(ridade_arv):
    for tarn in range(tarnide_arv):
        print("*", end="")
        # end="" panen print juurde sellepärast,
        # et igal tiirul kirjutataks tärn samale reale
    print() # reavahetus
```

Kindlasti katseta ülal toodud näidet igat pidi. Vaata mis juhtub kui muuta printide asukohti (nende taandeid) ja proovi leida vastus küsimusele miks. Muuda ka ridade arvu ja tärnide arvu. Uuri, mis siis saab, kui ma teise for ette taanet ei paneks jms.

Milleks on topelttsüklid mängude tegemisel head?

Paljude mängude, eriti nn lauamängudes kasutatake topelttsükleid päris palju. Mõned näited:

- Lauaplaat - nt malelaud, koosneb regulaarselt üksteise otsa ja kõrvale joonistatud ruutudest. Selliste jooniste tegemisel kirjutatakse vaid üks kord ruudu joonistamise käskude rida ja ülejäänud 64 korda ruudu joonistamist korraldatakse topelttsükliga. Täpselt sama moodi nagu tärnide joonistamiselgi, vaid ühele reale kirjutasime, et trüki tärn, kuid ekraanile tuli tärne ju palju, palju rohkem.
- Teine valdkond, kus topelttsükleid või isegi mitmekordseid tsükleid kasutatakse, on erinevate juhtumite läbivaatamine ehk kombinatsioonide ja permutatsioonide leidmine. Jällegi võib siinkohal tuua male näite, arvuti vaatab teatud sügavusele käikude võimalikud kombinatsioonid ette ja valib sealt kõige otstarbekama. Kombinatsioonidest ja permutatsioonidest räägime täpsemalt mõnes järgmises peatükis.

Mitmekordsed tsüklid

Teeme asja aga veel põnevamaks. Lisame veel tsükleid üksteise sisse. Eelmistes peatükkides lasime arvutil kirjutada ühe tsükliga (sisemine) tärne ühte ritta, kahe tsükliga (välimine ja sisemine) tekitasime palju ühesuguseid ridu. Kuid lisame veel ka kolmanda tsükli kõikide eelnevate ümber. Huvitav, mis nüüd küll juhtuma võiks hakata?

```
print("Mitu tärni kirjutan ühte ritta?")
tarnide_arv = int(input())
print("Mitu rida trükin?")
ridade_arv = int(input())
print("Mitu plokki ridu trükin?")
plokkide_arv = int(input())

for plokk in range(plokkide_arv):
    for rida in range(ridade_arv):
        for tarn in range(tarnide_arv):
            print("*", end="")
            # end="" panen print juurde sellepärast,
            # et igal tiirul kirjutataks tärn samale reale
        print() # reavahetus
    print() # tühi rida
```

Näide kilpkonnagraafikaga:

Kombinatsioonid ja permutatsioonid

Nagu eelnevalt korraks juttu oli, siis on mitmekordsed tsüklid head just kõikvõimalike kombinatsioonide ja permutatsioonide välja selgitamiseks.

Mis asjad on kombinatsioonid ja permutatsioonid?

- **Permutatsioon** on matemaatiline termin, mis tähendab unikaalset viisi, kuidas kombineerida asjade järjestust.
- **Kombinatsioon** on peaaegu sama asi, kuid siin elementide järjestus ei loe, ta ei ole unikaalne.

Vaatame mõningaid näiteid:

Kui ma palun sul valida välja kolm arvu 1-10, siis kaks võimalikku varianti oleksid järgmised

- 8, 3, 5
- 7, 2, 10

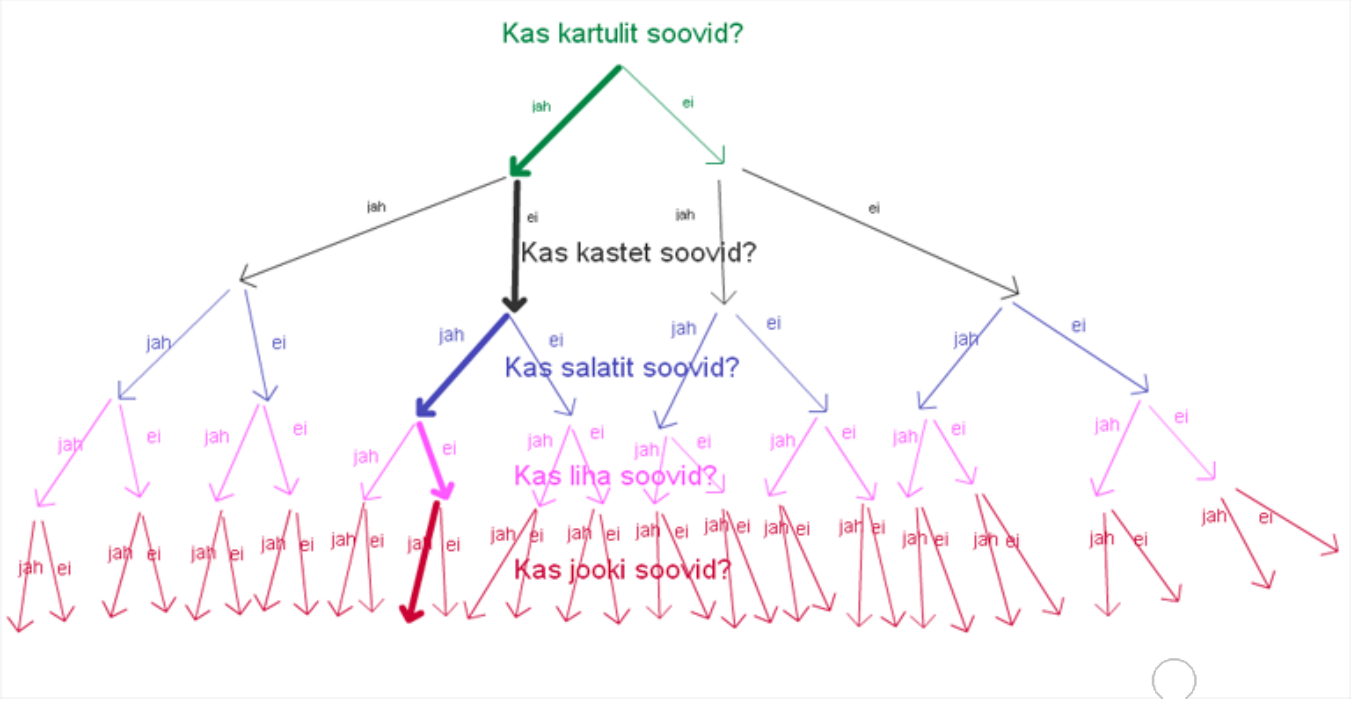
neid nimetatakse laiemalt kombinatsioonideks. Permutatsioon tuleb aga mängu siis, kui sa valisid näiteks kogemata samad arvud, kuid erinevas järjekoras

- 7, 2, 10
- 10, 7, 2
- 2, 7, 10

siis see on üks ja seesama arvude kombinatsioon, kuid kolm täiesti erinevat permutatsiooni.

Kõige parem viis nende mõistete seletamiseks, on tuua üks konkreetne näide. Olgu selleks kooli söökla. Oletame, et sul on võimalik valida oma sööklas päevaprae juures, mida sa taldrikule tahad panna ja kas sa jooki juurde võtad (komponentideks on üldjuhul midagi sellist: kartul, kaste, salat, liha, jook). Kuidas oleks võimalusega teada saada kõikvõimalikud erinevad kombinatsioonid, mida õpilased üldse võivad oma lõunasöögiks valida.

Üks võimalik viis sellist ülesannet lahendada, on kasutada otsustuste puud:



Sellel joonisel üks teekond tipust alla ongi üks kombinatsioon. Üks kõikidest võimalikest kombinatsioonidest on ka joonisel paksu joonega välja toodud. Sellel ühel juhul võttis õpilane kartulit, salatit ja joogi. Liha ja kastme jättis paremaid päevi ootama.

Teine võimalus selle ülesande lahendamiseks, on aga see ülesanne programmeerida:

```
print("\tkartul \tkaste \tsalat \tliha \tjook")
loendaja = 1
for kartul in [0, 1]:
    for kaste in [0, 1]:
        for salat in [0, 1]:
            for liha in [0, 1]:
                for jook in [0, 1]:
                    print(loendaja,"\t",kartul,"\t",
                          kaste,"\t",salat,"\t",liha,
                          "\t",jook)
                    loendaja +=1
```

Seda koodi uurides, võid imestada, et mis see "\t" seal igal pool teeb. Kuid kui sa koodi käima panid, siis paned tähele kui ilusasti on kõik andmed üksteise alla tabelisse reastatud. Seda rolli see "\t" mängibki - kirjutab kahe teksti vahele ühe tabi ehk tabulaatori.

0 ja 1 igas for-tsükli reas tähendab sama, mis jah ja ei otsustuste puus, nii et kui sulle 0 ja 1 ei meeldi, võid kirjutada ka "jah" ja "ei":)

Programmi tulemusest on näha, et sööklas oleks sellisel korral võimalik 32 erinevat tellimust teha, teine küsimus on muidugi see, kui mõttekad mõned kombinatsioonid neist on, aga nagu öeldakse, maitse üle ei vaielda.

Näide

Vaatame veelkord söökla näidet ja proovime selle teha veel huvitavamaks. Lisame iga toiduaine juurde ka tema kalorite hulga ja arvutame iga rea lõppu välja valitud portsioni kalorite summa. Programmis on kasutatud ligilähedasi kalorite suurusi, nii et need peaksid peaaegu isegi õiged olema:)

Kuidas seda ülesannet lahedada? Igale toiduliigile tuleb lisada kaloritega varustatud muutujad ja lisada kõige sisemisse tsüklisse ka kalorite kokku arvutamise valem. Proovi seda teha kõigepealt iseseisvalt ja alles siis vaata, kuidas olen selle ülesande lahendanud mina.


```
kartul_kal = 220
kaste_kal = 125
salat_kal = 20
liha_kal = 175
jook_kal = 90

print("\tkartul \tkaste \tsalat \tliha \tjook \tkalorid")
loendaja = 1
for kartul in [0, 1]:
    for kaste in [0, 1]:
        for salat in [0, 1]:
            for liha in [0, 1]:
                for jook in [0, 1]:
                    kalorid=(kartul*kartul_kal +
                             kaste*kaste_kal +
                             salat*salat_kal +
                             liha*liha_kal +
                             jook*jook_kal)
                    print(loendaja,"\t",kartul,"\t",
                          kaste,"\t",salat,"\t",liha,
                          "\t",jook,"\t",kalorid)
                    loendaja +=1
```

List

Mäletate, rääkisime kunagi andmetüüpidest. Vaatasime lähemalt arvulisi - *integer* ja *float* tüüpi ning sõnesid (*string*). Kui tahtsime mingit tüüpi andmeid mälus hoida, siis võtsime appi muutuja, millele omistasime "mingi asja". Kuid elus on palju selliseid olukordi, kus on vaja hoida ühes muutujas mitmeid elemente, näiteks

- nimekirja tegelase omadustest,
- kuude või nädalapäevade loetelu,
- sõprade nimekiri jne

Väga tüütu oleks kõiki neid mingi ühise tunnusega kokku kuuluvaid andmeid üksikult muutujates hoida. Sellepärast on programmeerimises kasutusel **listid**. List on praeguseks neljas andmetüüp, mida vaatame. Listidega puutusime juba natuke kokku for-tsükli uurimisel, kuid selles peatükis räägime neist veidi täpsemalt - vaatame kuidas liste luuakse, kuidas neid muudetakse ja kasutatakse.

Mis on list?

List on grupp või kollektsioon kokku kuuluvatest, ühist nimetust omavatest andmetest. Andmed ühes listis võivad olla mistahes tüüpi - tekstid, arvud, isegi teised listid. Tavaliselt aga üritatakse ühes listis hoida ühte tüüpi andmeid. Näiteks nimekiri toiduainetest või kogunenud aaretest.

Kuidas listi luuakse?

Nagu kõikide andmetüüpide korral Pythonis, võetakse kõigepealt kasutusele uus muutuja, millele omistatakse uue andmetüübi sisu. Arvude korral tuli võrdusmärgi taha kirjutada lihtsalt arv või komaga arv, sõne moodustamiseks pidi väärtusele ümber panema jutumärgid. Listi korral aga tuleb andmete loetelu panna **kandilistesse sulgudesse**.

```
>>> värvid = ["roheline", "punane", "kollane", "sinine"]
>>> kinganumbrid = [36, 37, 38, 39, 40, 41, 42]
>>> linnad = ["Tartu", "Pärnu", "Türi", "Narva"]
>>> hinded = [2, 3, 4, 5]
>>> punktid = [45.6, 67.8, 56.7, 34.6]
```

Nii saab luua juba nö valmis ehk täis listi, kuid võib alguses luua ka täiesti tühja listi ja siis hakata sinna elemente juurde lisama. Elemente saab juurde lisada tegelikult mistahes listidesse, nii tühjadesse või juba nö valmis listidesse, kuid meeles tuleb pidada seda, et **lisatakse** alati listi kõige lõppu **viimasele kohale**.

Järgmine pilt illustreerib, kuidas kõigepealt luuakse tühi list (tühjad nurksulud). Siis lisatakse listi käsu `.append()` abil listi uus element. Ning kui nüüd list välja trükkida, siis näeme, et listi on lisandunud üks liige. **NB!** listi sees ei kao erinevad andmetüübid mitte kusagile. Kui listis tahetakse hoida sõnesid, siis tuleb neile jutumärgid ümber panna ja kui listis tahetakse hoida teisi liste, siis tuleb selle määramiseks kõikidele listi elementidele panna omakorda nurksulud ümber. Ava käsurea keskkond ja tee järgmine näide läbi ja kindlasti rakenda seal juures ka oma fantaasiat ning vaata, kuidas Python sellele reageerib:

```
>>> sõbrad = []
>>> sõbrad.append("Jüri")
>>> print(sõbrad)
['Jüri']
>>> sõbrad.append("Mari")
>>> print(sõbrad)
['Jüri', 'Mari']
>>>
```

Pea meeles, et kui listi hakata midagi lisama, peab sul enne list defineeritud olema, nagu on seda tehtud ülemisel pildil. Analoogia võid leida näiteks koogitegemisest, ei saa ju lihtsalt kõik koogitaigna jaoks vajalikud ained kokku valada, enne kui sa pole kapist välja võtnud kaussi, kuhu taigent panna!!!

Listist elemendi välja võtmine

Üks võimalus listist elemente kätte saada, on tema järjekorra numbri järgi. **NB!** Ole hoolikas! Nagu kõikide asjadega siin Pythonis, igasuguste nimekirjade, listide jms järjekorranumbreid hakatakse loendama **nullist**. Uuri järgmist pilti ja proovi listist elementide kätte saamist ka oma Idles.

```
>>> linnad = ["Tartu", "Pärnu", "Türi", "Narva"]
>>> print(linnad[2])
Türi
>>> print(linnad[0])
Tartu
>>> print(linnad[4])
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    print(linnad[4])
IndexError: list index out of range
>>>
```

Üleval näites tuli veateade, kuna selles listis ei ole viite elementi. linnad[4] tähendab listi viiendat elementi!

Listi saab jaotada ka väiksemateks listideks, kuid selle erandiga, et vastavad andmed peavad listis paiknema järjest:

```
>>> linnad = ["Tartu", "Pärnu", "Türi", "Narva"]
>>> print(linnad[0:2])
['Tartu', 'Pärnu']
>>> print(linnad[1:3])
['Pärnu', 'Türi']
>>> print(linnad[2:4])
['Türi', 'Narva']
>>> print(linnad[-1])
Narva
>>>
```

Pane tähele, et kasutada võib ka negatiivseid arve, nii saad kõige kiiremini kätte listi kõige viimase elemendi. Kuidas saaks kiiresti eelviimase elemendi?

Listi elemendi muutmine

Samuti listi järjekorranumbri järgi saab muuta ka sellel positsioonil oleva elemendi väärtust listis. Seda tehakse nii:


```
>>> linnad = ["Tartu", "Pärnu", "Türi", "Narva"]
>>> linnad[0]="Paide"
>>> print(linnad)
['Paide', 'Pärnu', 'Türi', 'Narva']
>>>
```

Listi lisamine 2

Peale `.append()` käsu on veel võimalusi Pythonis, kuidas listi elemente juurde lisada.

`.extend()` käsuga lisatakse terve teise listi sisu esimesele listile juurde st ühendatakse kaks listi omavahel.

`.insert()` käsuga saab listi konkreetsele kohale uue elemendi juurde panna st vahele sokutada.

```
>>> linnad = ["Tartu", "Pärnu", "Türi", "Narva"]
>>> linnad[0]="Paide"
>>> print(linnad)
['Paide', 'Pärnu', 'Türi', 'Narva']
>>> linnad.extend(["Tartu", "Võru", "Rakvere"])
>>> print(linnad)
['Paide', 'Pärnu', 'Türi', 'Narva', 'Tartu', 'Võru', 'Rakvere']
>>> linnad.insert(2, "Jõgeva")
>>> print(linnad)
['Paide', 'Pärnu', 'Jõgeva', 'Türi', 'Narva', 'Tartu', 'Võru', 'Rakvere']
>>>
```

Ära unusta, et listides hakkab loendamine 0-st, mis tähendab et kohale 2 elemendi lisamine tähendab tegelikult kolmandat positsiooni listis.

Listi elemendi kustutamine

Kuidas saab listi elementi listist eemaldada? Selleks on kolm võimalust **`.remove()`**, **`del`** ja **`.pop()`**. Mis neil vahet on?

- **`.remove()`** kustutab elemendi tema nime järgi
- **`del`** võtmesõna kustutab indeksi järgi
- **`.pop()`** eemaldab listist viimase elemendi, kuid teda võib kasutada ka indeksiga koos, mis paneb ta töötama sama moodi nagu **`del`** võtmesõna.

```
>>> linnad
['Võru', 'Türi', 'Tartu', 'Rakvere', 'Pärnu', 'Paide', 'Narva', 'Jõgeva']
>>> linnad.remove("Tartu")
>>> linnad
['Võru', 'Türi', 'Rakvere', 'Pärnu', 'Paide', 'Narva', 'Jõgeva']
>>> del linnad[2]
>>> linnad
['Võru', 'Türi', 'Pärnu', 'Paide', 'Narva', 'Jõgeva']
>>> linnad.pop()
'Jõgeva'
>>> linnad
['Võru', 'Türi', 'Pärnu', 'Paide', 'Narva']
>>> linnad.pop(2)
'Pärnu'
>>> linnad
['Võru', 'Türi', 'Paide', 'Narva']
>>>
```

Listist otsimine

Tore on küll listis igasuguseid andmeid hoida, aga kui meil peaks andmeid vaja olema, kuidas ma leian need listist üles. Kuidas ma saan üldse kindlaks teha, et väärtus, mida ma otsin, samuti antud listis on?

Väga tihti tuleb listidega teha kahte asja:

- leida, kas mingi element on antud listis või mitte
- leida, millisel kohal minu element listis paikneb

Võtmesõna *in*

Vastame kõigepealt esimesele küsimusele, kas minu element on listis?

```
>>> linnad
['Paide', 'Pärnu', 'Jõgeva', 'Türi', 'Narva', 'Tartu', 'Võru', 'Rakvere']
>>> if "Pärnu" in linnad:
        print("jah")
else: print("ei")

jah
>>> "Pärnu" in linnad
True
>>> "Keila" in linnad
False
>>>
```

Nagu pildilt näha võib, annab ***in*** võtmesõna kasutamine alati, kas *True* või *False* vastuseks. Seda *Trued* ja *False* saab ära kasutada *if*-lauses, kui on tõsi prindi jah, kui ei ole tõsi prindi ei.

Elemendi järjekorranumbri ehk indeksi leidmine

Otsitava elemendi järjekorranumbri ehk **indeksi** saame kätte `.index()` käsuga:

```
>>> linnad
['Paide', 'Pärnu', 'Jõgeva', 'Türi', 'Narva', 'Tartu', 'Võru', 'Rakvere']
>>> linnad.index("Pärnu")
1
>>> if "Tartu" in linnad:
        print(linnad.index("Tartu"))


5
>>>
```

Listi sorteerimine

Listides on elemendid alati sellises järjekorras nagu need on sinna lisatud ja igal listis oleval elemendil on oma kindel indeks ehk järjekorranumber. Järjekorda saab muuta vaid **`.insert()`**, **`.append()`**, **`.remove()`** või **`.pop()`** käskudega. Kuid see järjekord ei pruugi olla kõige meelepärasem. Sageli oleks vaja, et listi liikmed oleksid sorteeritud, näiteks arvud kõik reastatud kasvavalt või tekstid kõik tähestikulises järjekorras.

Listide sorteerimiseks kasutatakse käsku **`.sort()`**

```
>>> linnad
['Paide', 'Pärnu', 'Jõgeva', 'Türi', 'Narva', 'Tartu', 'Võru', 'Rakvere']
>>> linnad.sort()
>>> linnad
['Jõgeva', 'Narva', 'Paide', 'Pärnu', 'Rakvere', 'Tartu', 'Türi', 'Võru']
>>> punktid
[45.6, 67.8, 56.7, 34.6]
>>> punktid.sort()
>>> punktid
[34.6, 45.6, 56.7, 67.8]
>>>
```

Saab sorteerida ka vastupidises järjekorras käsuga **`.reverse()`**

```
>>> linnad
['Jõgeva', 'Narva', 'Paide', 'Pärnu', 'Rakvere', 'Tartu', 'Türi', 'Võru']
>>> linnad.reverse()
>>> linnad
['Võru', 'Türi', 'Tartu', 'Rakvere', 'Pärnu', 'Paide', 'Narva', 'Jõgeva']
>>> punktid
[34.6, 45.6, 56.7, 67.8]
>>> punktid.reverse()
>>> punktid
[67.8, 56.7, 45.6, 34.6]
>>>
```

Mida õppisid?

Sellel nädalal sai läbi töötatud programmeerimise vaatepunktist väga olulised konstruktsioonid. Seega kui sa need teadmised endale selgeks tegid ja praktikas neid palju katsetad, siis peaaegu polegi enam väga palju rohkem vaja. Siiani omandatud teadmiste oskuslikul kombineerimisel peaksid oskama juba päris vingeid asju teha. Kuid vaatame veelkord üle, mis see nädal selgeks sai:

- mis asjad on topelt- ja mitmekordsed tsüklid?
- milliseid huvitavaid asju saab topelttsüklitega teha?
- miks on topelttsüklid asendamatud?
- mis on permutatsioonid ja kombinatsioonid?
- kuidas on kombinatsioonid seotud mitmekordsete tsüklitega?
- mis on otsustustepuu?
- mis on list?
- kuidas liste luua?
- kuidas listi elemente juurde lisatakse ja ära võetakse?
- kuidas listist elemente kustutatakse?
- kuidas teha kindlaks, kas element asub listis?
- kuidas teha kindlaks, kus element listis asub (tema indeks)?
- kuidas listist elementi kätte saada?
- kuidas saada kiiresti kätte listi viimane element?
- kuidas liste sorteerida?

TEE ISE!

Keerulise teemaga nädal. Teoorias ei pruugi see kohe nii paistagi, kuid asu kibekiiresti ülesannete juurde ja kontrolli, kas oled saanud topelttsüklitest ja listidest just nii aru nagu võiks.

1. Ülesanne: Kujundid. Kirjuta programm, mis väljastab ekraanile pildil näha olevad kujundid (iga tärni vahel reas on üks tühik). Esimene kujund on alati sümmeetriline, st ridu on sama palju kui veerge.

```
Sisesta kujundi ridade arv:5
* * * * *
*       *
*       *
*       *
* * * * *

*
* *
* * *
* * * *
* * * * *
```

Kujundi ridade arv tuleb küsida programmi alguses kasutajalt.

2. Ülesanne: Kilpkonn.

```
from turtle import *

#järgmine tsükkel joonistab ruudu
for i in range(4):
    forward(50)
    left(90)

# forward tähendab kilpkonna edasi liikumist
# 50 tähendab 50 pikselit
# left tähendab pööramist vasakule
# 90 tähendab 90 kraadi
# up() tõstab pliiatsi üles, jälge ei tehta
# down() paneb pliiatsi alla, saab joonistada
# home() viib kilpkonna alguspunkti, mis asub ekraani keskel
```

turtle on nn kilpkonnagraafika moodul, kus kolmnurgaga saab teha ekraanile lihtsaid graafilisi joonistusi kilpkonna käskudega, mõned käsud on toodud kommentaarides. Kõikvõimalikud käsud, kui neid peaks vaja minema, leiad turtle mooduli dokumentatsioonist veebis.

Sinu ülesanne on selle ühe tsükli abil joonistada mängulaud, mis on 8x8 ruuduga. Peab kasutama topelttsüklit.

3. Ülesanne: Liitmine. Koosta programm, mis arvudega täidetud listi igale elemendile liidab juurde kasutajalt küsitud arvu (list loo ise). Näiteks alguses on list `[14, 3, 25]` ja peale 3 liitmist on list `[17, 6, 28]`. Programm peab mõlemad listid kirjutama ekraanile originaalkujul ja ka sorteerituna.

4. Ülesanne: Kombinatsioonid. Mõtle ühele elulisele olukorrale (u nagu minu söökla näide), kus oleks

põnev teada saada kõikvõimalikud erinevad kombinatsioonid. Realiseeri olukord programmina ja väljasta tulemus tabelkujul.

5. Ülesanne: Tagasiside. Vasta tagasiside lingi alt mõnedele küsimustele, et mul oleks ülevaade sinu edasijõudmistest.