Sisukord

RAAMAT 2

Otsustamine

Tingimusekontroll
- Mitme tingimuse kontroll 1
- Mitme tingimuse kontroll 2

Tsükkel
- FOR tsükkel
- Lõpmatu tsükkel
- RANGE()
- Tsüklimuutujad
- Tsükkel sõnedega
- AJAMÕÕTJA!
- WHILE tsükkel
- Jäta tsükkel pooleli!

Kommentaarid

Mäng "Arva arvu" uuesti

Kuidas kavandada mängu?

Mida õppisid?

TEE ISE!

Kursuse "Teeme ise arvutimänge - algus"

2. RAAMAT

OTSUSTUSED, HARGNEMINE JA TSÜKLID

Tiina Kull

Tartu Ülikool

2012

Otsustamine

Praeguseks oleme selgeks saanud elementaarmõisted, oskame arvutit sundida väljastama sõnumeid ja oskame teha nii, et arvuti küsiks ise kasutajalt, failist või lausa veebist andmeid. Oskame panna arvutit arvutama meie eest ja oskame salvestada info arvutimällu.

Ükski tänapäeva programm ilma **otsustusvõimeta** ei kvalifitseeru programmiks. See on üks olulisemaid tingimusi programmide juures üldse - ta peab olema nö "mõtlemisvõimeline", olema suuteline tegema otsuseid ja vastavalt otsustele ka tegutsema. Eelmise nädala teemad on kõik väga vajalikud teadmised, mida on tarvis programmi kirjutamiseks ja eriti mängu kirjutamiseks. Kuid sellisel sisend-väljund-muutuja tasemel programmi või mängu võib võrrelda pigem kivikirvega kui näiteks AngryBirds'ga. Seega, et jõuda selliste teadmisteni, et teha Angry Birdsi-taoline mäng, peame veel palju teooriat KOOS praktikaga omandama. See, kui hästi mõni mäng töötab, oleneb sellest, kui hea on olnud selle mängu programmeerija. Hea programmeerija aga kujuneb AINULT läbi tohutul hulgal programmeerimise, ükskõik kui väikesed või suured need programmid ka alguses ei oleks. Selle nädala kaks suurt põhiteemat ilma milleta ei valmi ükski normaalne programm ega sirgu ükski programmeerija, on **tingimuslause** ja **tsükkel**.

Aeg panna oma programmid tegema otsuseid.

Programm peab oskama vastavalt sisendile käituda erinevalt. Näiteks:

- Kui Juku valis ekraanilt õige aarde, siis tuleb tema skoorile üks pall juurde lisada.
- Kui Kati saab vigastada, tuleb tal üks elu maha võtta.
- Kui Jack sai vastasele pihta, tuleb teha plahvatuse häält.
- Kui otsitavat faili ei leita, siis tuleb anda veateade.
- Kui Mikk jookseb vastu kasti, siis kast haihtub vms.

Otsuste tegemiseks on eelnevalt vaja kontrollida **tingimust**. Ehk kõigepealt kontrollime, kas Jack sai vaenlasele pihta ja alles siis paneme vastava *wav*-faili mängima:)

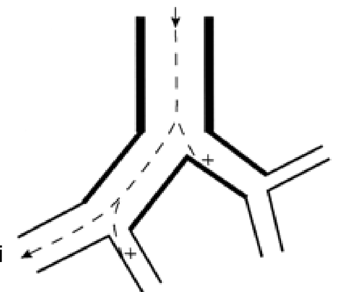
Oluline on aru saada, et tingimuse kontrolli tulemus saab olla AINULT õige või vale (i.k ***true*** or ***false***). Kas mingi asi on toimunud või ei ole toimunud?

Arvutile aga ei saa lihtsalt öelda, et hei, Jack sai vaenlasele pihta, tee plahvatuse häält! Arvuti on rauast ja oskab aru saada vaid kahendkoodist ehk siis ainult arvudest:(

Toon sulle mõned näited, millistele küsimustele saab arvuti vastata:

- Kas kaks muutujat on võrdsed?
- Kas üks muutuja on teisest suurem?
- Kas üks muutuja on teisest väiksem?

Lisaks, ei ole kahe muutuja omavaheline võrdlemine kah nii sama. Muutujad, mida omavahel võrreldakse, peavad olema **ühte tüüpi.**



Seega, peab programmeerija kõik vaimustavad vaenlased ja supersõdalaste kõik liigutused ümber kodeerima arvudeks, mida saab omavahel võrrelda ja alles seejärel saab teha otsuseid. Seega võiksime Jacki loo ümber tõlkida umbes nii:

> Kui Jack tabas vibunoolega vaenlast, siis muutuja *vaenlane* väärtus muudetakse nulliks ning tingimuse kontrollija kontrollib samal ajal, et kui `vaenlane=0`, siis pannakse mängima plahvatuse heli. Kui mitte, siis mingit plahvatust ei tule.

Sellist ühes või teises suunas otsuste tegemist ja liikumist vastavalt tingimuse kontrollile nimetatakse programmeerimises ka **hargnemiseks**. Programm otsustab, millist haru pidi edasi minnakse, kui tingimus on üks, keerab näiteks vasakule, kui mitte, keerab paremale.

Tingimusekontroll

IF-konstruktsioon

Vaatame täpsemalt, kuidas arvutile tingimuste kontroll selgeks teha.

Pythonis pannakse tingimuste kontroll kirja võtmesõna **IF** kaudu (võtmesõnad on Pythoni Idle-s tavaliselt oranžikas-kollased):

```
if JukuVastus == õigeVastus :
    print("Tubli! Õige vastus")
    skoor += 1
print("Aitäh mängimast!")
```

Kõrvalolevas koodijupis kontrollitakse, kas Juku vastus on sama, mis õige vastus, kui on, siis täidetakse if-lause järel tulev plokk. Kui mitte, siis trükitakse mängijale lihtsalt "Aitäh mängimast!".

Mis on plokk? **Plokk ehk tingimuslause keha** moodustavad kõik need if-lausele järgnevad koodiread, mis on **taandatud**, ehk siis antud näites kaks rida peale koolonit. Taane on klassikaliselt **4 tühikut**, kuid võid kasutada ka kahte või viite vms arvu tühikuid. Oluline on see, et ühe programmilõigu raames kasutaksid ühesugust taande pikkust, vastasel korral saad veateate. Samuti on väga oluline **koolon** if-lause lõpus, koolon annabki Pythonile märku sellest, et nüüd on tulemas if-lause keha. Kui tingimus on õige, täidetakse kõik tingimuslause keha käsud, kui mitte, jäetakse keha plokk vahele ja minnakse järgmiste käskude juurde.

Kas ma näen topelt?

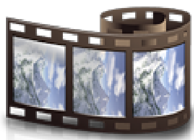
Kas tõesti on võrduse kontrollimisel kasutatud kahte võrdusmärki? Jah, on küll. Selle pead endale väga täpselt selgeks tegema, et kui on vaja kontrollida kahe muutuja või arvu võrdsust, siis tehakse seda **topeltvõrdusmärgiga** (**==**). Miks? Sest ühekordset võrdusmärki kasutatakse muutujatele andmete omistamisel. Muutujale väärtuse andmine ja võrduse kontroll on sellisele rauast kastile nagu arvuti kategooriliselt erinevad asjad, seepärast tuleb kasutada ka erinevaid märke.

= ja **==** segamini ajamine on programmeerijate (ka juba kogenud) üks kõige tavalisemaid vigu. Väga paljud programmeerimiskeeled kasutavad samasugust sümboolikat ja väga paljud programmeerijad kasutavad vale märki vales kohas. Ole siis hoiatatud!

Teisi kontrolli võimalusi

Lisaks kahe muutuja võrdsusele võib loomulikult if-lauses kasutada ka mistahes teisi võrdlemise sümboleid.

võrdlusoperaator koos näitega	tähendus
`if arv1 == arv2 :`	Kas kaks muutujat on võrdsed?
`if arv1 > arv2 :`	Kas arv1 on suurem kui arv2?
`if arv1 < arv2 :`	Kas arv1 on väiksem kui arv2?
`if arv1 >= arv2 :`	Kas arv1 on suurem või võrdne arvuga 2?
`if arv1 <= arv2 :`	Kas arv1 on väiksem või võrdne arvuga 2?
`if arv1 != arv2 :`	Kas kaks muutujat on erinevad?
`if 20 > arv > -5 :`	Kas arv jääb -5 ja 20 vahele?
`if -5 <= arv <= 20 :`	Kas arv jääb lõiku -5 kuni 20?



Mitme tingimuse kontroll 1

Tingimuse kontrolli saab teha palju, palju huvitavamaks. Sageli ei taha ma kontrollida ainult ühte asja vaid mitut. Lisaks tahan ma, et arvuti vastavalt erinevatele tingimustele ka erinevalt käituks. Sellist asja saab teha **IF-ELIF-ELSE** kontruktsiooniga.

IF-ELIF-ELSE konstruktsioon

Selle konstruktsiooni selgitamiseks võtan appi ühe võimaliku jupi seiklusmängust. Mängud koosnevad pea alati paljudest erinevatest alamprogrammidest, mis teevad erinevaid asju. Üheks selliseks alamprogrammiks võib olla justnimelt mingi otsustuse kontroll. Vaatame seda lähemalt:

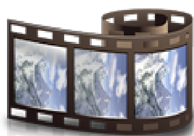
```
print("""Hei, seikleja!
Oled oma rännakul jõudnud teede ristumiskohta.
Millist teeharu pidi kavatsed edasi minna?
Kas paremale (p), vasakult (v) või otse (o)?""")
otsus = input()
skoor = 0
if otsus == "p":
    print("""Otsustasid minna paremale,
see tee viib sind kolme peaga lohe juurde!""")
    skoor -= 1
elif otsus == "v":
    print("""Otsustasid minna vasakule,
see tee viib sind kuningalossi!""")
    skoor += 1
elif otsus == "o":
    print("""Otsustasid minna otse,
see tee viib sind koju""")
    skoor += 1
else :
    print("Sa ei valinud õiget tähte seiklemiseks!")
print("sa kogusid",skoor, "punkti")
```

Kõigepealt antakse mängijale teada, et ta on teelahkmel ja oma edasise tee valimiseks tuleb tal vajutada p, o või v tähte. Mängija valik salvestatakse muutujasse `otsus`. Samuti on kasutusel veel üks muutuja `skoor`, mis on võrdsustatud alguses nulliga. Seejärel hakkab toimuma otsustuse analüüs ehk kontroll.

- Kui `otsus` on võrdne p-ga, täidetakse esimese if-lause plokk (plokk oli siis if-lause ja kooloni järel tulev TAANDATUD käskude loetelu)

Ole siin tähelepanelik, muutujate võrdlemisel peavad muutujad olema **sama tüüpi**. Kuna otsus on väärtustatud `input()` käsuga, siis on ta sõne tüüpi ja selle muutuja võrdlemiseks peab samuti kasutama sõnet ehk tähele "p" peab kindlasti jutumärgid ümber panema.

- Kui esimene tingimus aga täidetud ei ole, siis minnakse kohe järgmise võtmesõna **ELIF** juurde ja kontrollitakse sealset tingimust. Kui ka see ei osutu tõeks, võetakse järgmine. Selliseid ELIF-e võid sa kirjutada üksteise alla nii palju kui soovid, kuid sageli osutub liiga paljude ELIF-de kasutamine mitteökonoomseks koodi kirjutamiseks. Kuidas selliseid olukordi vältida, sellest räägime aga mõnes järgmises peatükis.
- Eriline on viimane kontroll! Kui kõik eelmised tingimused on kontrollitud ja ükski ei vastanud tõele, siis on mõistlik lisada **ELSE** osa, mille järel ei tule ühtegi tingimuse kontrolli. See on kõikide muude juhtude jaoks (näiteks juhu jaoks, kus mängija on kirjutanud vale tähe vms.)
- Ning kõige viimaks täidetakse antud programmis skoori välja printimise rida, mis ei sõltu enam ühestki eelmisest if- või elif-lausest



Mitme tingimuse kontroll 2

Veel keerulisemate olukordade jaoks on vaja teha mitu tingimuse kontrolli ühes IF-lauses. Mida see tähendab?

Kui enne kontrollisime iga IF- või ELIF-lause juures ühte tingimust, näiteks kas `otsus == "p"`, siis tegelikult saab ühe IF- või ELIF-lause juures teha mitu kontrolli korraga.

AND ja OR ja NOT

Jällegi võtan appi ühe võimaliku stsenaariumi jupi mängu loomisel asja selgitamiseks.

```
print("Kui vana sa oled?")
vanus = int(input())
print("Mitmendas klassis sa käid?")
klass = int(input())

if vanus > 18 or klass > 11:
    print("Sa oled selle mängu mängimiseks liiga vana!")
elif vanus < 10 and klass < 4:
    print("Oota veel natuke")
else:
    print("Alusta mängimist!")
```

Kõigepealt tahab programm teada kasutajalt kahte asja, tema vanust ja tema klassi. (Eeldame, et need sisestakse korralikult arvulistena, mitte ei kirjutata teksti)

- Pane tähele, et muutujatele `vanus` ja `klass` väärtuste omistamisel muudetakse ka andmetüüpi. Input() annab sõne, aga me muudame int() käsuga sõne täisarvuks, siis on hiljem lihtsam võrrelda.
- Tingimuste kontrollis esimeses if-lauses kontrollitakse kahte asja, et vanus ei oleks üle 18 **VÕI** klass ei oleks üle 11. Mida see tähendab? St seda, et kui mängija sisetab kas ühe või teise väärtuse liiga suure, näiteks vanuse 17, aga klassi 12, siis antakse talle teade, et ta on selle mängu jaoks liiga vana:(
- Elif-lauses kontrollitakse samuti kahte asja aga seekord veidi teise seosega. Nimelt siinses kontrollis saab mängukeelu AINULT siis, kui mõlemad tingimused on täidetud: `vanus` peab olema alla 10 **JA** `klass` peab olma alla 4. Seega mängija, kes on näiteks 9 aastat vana, aga käib 5. klassis, saab seda mängu mängida.

Kui on soov kontrollida, et üks tingimus kehtiks ja teine tingimus kindlasti mitte ei kehtiks, siis tuleb kasutada AND või OR koos võtmesõnaga **NOT**. Näiteks: **`arv1>10 and not arv==20`**

Tsükkel

Tsükli peatükid on programmeerimise õppimise juures üliolulised. Seega, kus vähegi näed Tee ise! logo, siis nüüd küll on viimane aeg tõepoolest katsetada neid näiteid ka oma arvutis. Tsüklite olemusest lihtsalt peab aru saama ja niisama lugemine siin ei aita.

Ühe ja sama asja tegemine tuhandeid kordi järjest on inimesele väga kurnav ja tüütu, kuid arvutile on sellised ülesanded kui loodud. Nad võivad ühte ja sama asja (näiteks kirjutada ekraanile "krokodill" tuhat korda) teha murdosa sekundiga. Mitu sekundit võtaks sinul "krokodill" kirjutamiseks?

Selles peatükis vaatamegi, kuidas panna arvutit kordama ehk arvutikeeles, kuidas moodustatakse **tsükleid**.

Tsüklid jagunevad kaheks liigiks:

- Tsüklid, mis kordavad teatud koodijuppi kindel arv kordi - selliseid tsükleid kutsutakse **FOR-tsükliteks**.
- Tsüklid, mis kordavad oma sisu nii kaua, kuni määratud tingimus on tõene, selliseid tsükleid nimetatakse **WHILE-tsükliteks**.

Vaatame neid lähemalt!

FOR tsükkel

Ava Idle tekstiredaktori aken **File > New** ja kirjuta sinna järgmised kaks rida, salvesta (näiteks nimega *tsykkel1.py*) ja vajuta **F5**.

```
for loendaja in [1, 2, 3, 4, 5]:
    print("krokodill")
```

Hei, need kaks rida koodi kirjutasid "krokodill" viis korda ehkki print on kirjutatud ainult ühe korra!

Justnimelt! Selles see võlu ongi:

- esiteks suudab arvuti teha midagi väga palju kordi väga lühikese ajaga
- teiseks hoiab selline kirjaviis tohutult kokku programmi kirjutatavate ridade arvu. Viie print() rea asemel kirjutasime kaks rida for-tsükli abil.

Mõned selgitused, kuidas for-tsükkel töötab.

Mõisted:

- sõna `loendaja` on **tsüklimuutuja.** Tsüklimuutuja saab igal kordusel uue väärtuse tema taga olevast nimekirjast. Muutuja nime võib ise valida.
- `in` on võtmesõna, mis ütleb muutujale, kust väärtused tuleb võtta.
- [1, 2, 3, 4, 5] on **list** ehk nimekiri. List kirjutatakse ALATI kandilistesse sulgudesse. Listi sisu võib vaadata kui tsüklimuutuja väärtuste järjekorda.
- **kooloni** järel tuleb taandega for-tsükli **keha** ehk koodi plokk, mida iga korduse korral täidetakse täies mahus. Antud näites on kehas ainult üks käsk, print().

Kuidas tsükkel töötab?

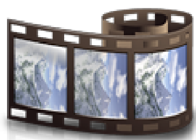
- For-tsükkel teeb täpselt nii mitu kordust, kui palju on tal väärtusi võtmesõna taga olevas listis. Antud juhul on seal 5 elementi, seega täidab tsükkel oma keha ehk print() käsku 5 korda.

Tegelikult, kui me tahaksime kirjutada "krokodill" 5 korda, siis ei oma listi sisu mingit tähtsust, **oluline** on vaid see, et listis oleks 5 elementi. Proovi näiteks kirjutada listi sisuks [1, 1, 1, 1, 1] või ['konn', 'karu', 'kass', 'madu', 'lind'].

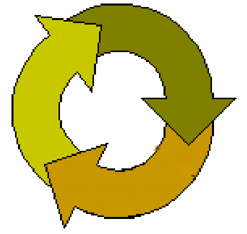
Proovime nüüd aga midagi veidi huvitavamat kui "krokodill". Ava uus aken, kirjuta järgmised read, salvesta ja pane käima:

```
for loendaja in [1,2,3,4,5]:
    print(loendaja)
```

Nagu tulemusest ka ise näed, kirjutab tsükkel seekord ekraanile tsüklimuutuja väärtused. Nagu ka eelmises näites, proovi tsükli listi sisuks panna ka midagi muud kui loetelu, näiteks [1, 1, 1, 1, 1] või ['konn', 'karu', 'kass', 'madu', 'lind'].



Lõpmatu tsükkel



Nagu nimest võib aimata, on lõpmatu tsükkel selline tsükkel, mis töötab lõpmatult:) Selline olukord tekib tavaliselt kogemata, tsükli kirjutamisel on tehtud viga nii, et tsükkel jääbki käima. For-tsüklitega juhub seda harva. While-tsüklites tihemini. Kuid ole mureta, pea igal programmeerijal juhtub selliseid asju aeg-ajalt, see on osa programmeerimisest.

Mida lõpmatu tsükliga teha?

Lõpmatult tööle jäänud tsükkel tuleb peatada. Peatamiseks tuleb vajutada **Ctrl+c** ja ongi tsükkel katkestatud.

RANGE()

Proovime for-tsükliga teha midagi kasulikku. Näiteks korrutustabelit.

Ava uus Idle aken, kirjuta sinna järgmised read, salvesta laiendiga *.py* ja pane käima.

```
for loendaja in [1,2,3,4,5]:
    print(loendaja, "x 7 on", loendaja*7)
```

Väga hea, kui sa selle koodijupi käima panid, siis ma arvan, et said põhimõttest aru, mida antud kaks rida tegid. Kuid kas see ei jäänud mitte veidi lahjaks? 5 rea korrutustabeli loomine ei ole ju mingi kunsttükk. Arvuti võimsusest teha asju tuhandeid kordi pole siin pea murdosagi kasutatud. Ma tahan, et minu programm arvutaks kõik korrutised 7-ga kuni tuhandeni!

Eelmises näites, me tegime seda aga ainult 5 korda: [1, 2, 3, 4, 5]

Kui ma tahan, et tsükkel kordaks ennast tuhat korda, kas ma pean siis listi kirjutama 1000 arvu?

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ...] ohh-oh-oh.... lol

Kindlasti mitte! Selleks programmeerimine ongi, et ei peaks mitte kusagil tegema mõttetut kordavat tööd ning sellisteks juhtudeks on kirjutatud Pythonisse sisse abifunktsioon `range()`.

Range() funktsioon annab listi, mis algab alati nulliga ja lõppeb üks arv enne sulgudesse kirjutatud arvu. Seega tuhandeni kirjutatud korrutustabel tuleb kirjutada nii:

```
for loendaja in range(1001):
    print(loendaja, "x 7 on", loendaja*7)
```

Kui ma peaksin tahtma aga mingit vahemikku, näiteks kõiki korrutisi 20-70ni, siis tuleb teha nii:

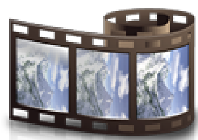
```
for loendaja in range(20,71):
    print(loendaja, "x 7 on", loendaja*7)
```

Range() funktsiooni sisse kirjutati algus ja lõpp, kusjuures tuleb meelde jätta, et lõpp tuleb kirjutada ühe võrra suuremana, kui tahta, et ka see arv oleks kaasaarvatud.

Range() funktsioonil on aga veel huvitavaid omadusi: proovi järgmist koodi!

```
for loendaja in range(1, 10, 2):
    print(loendaja, "x 7 on", loendaja*7)
```

Seekord lisasime range() funktsioonile kolmanda argumendi (nii nim. sulgudesse pandud arve). Nimelt sulgudes kolmandal kohal olev arv ütleb range() funktioonile kui mitmes väärtus sellest listist tuleb kasutusele võtta. Kuna näites oli kirjutatud 2, siis võeti korrutustabeli moodustamiseks iga teine arv. Samuti võib võtta iga 3nda või iga 5nda vms.



Tsüklimuutujad

Tsüklimuutujad on täpselt samasugused muutujad nagu kõik teisedki, need on abimehed, kes aitavad arvuti mällu teavet meelde jätta, ainult et antud juhul kasutatakse neid muutujaid tsüklite sees. Ja nagu muutujate peatükis oli juttu võib muutuja nimeks olla mistahes nimi, peaasi, et see vastaks Pythoni tingimustele. Kuid nii lihtsalt ma tegelikult ei pääse. Et kõik ausalt ära rääkida, siis pean siinkohal tunnistama, et tegelikult ma ei kasutanud tsüklite tutvustamisel eelmistel lehekülgedel üldlevinud muutujate nimesid. Nimelt on programmeerijatel tava, et tsüklimuutujatena kasutatakse üldjuhul tähti **i, j, k** jne. Miks nii?

Selline tava on välja kujunenud sellest, et algusaastatel, kui alles programmeerima hakati, olid arvutite ressursid väga piiratud ja iga täht oli nö arvel. Muutujate nimed olid võimalikult lühikesed. Arvuteid kasutati peaasjalikult matemaatiliste tehete tegemiseks. Matemaatilistes avaldistes nagu sa tead, kasutatakse palju tähti. Nii olidki tähestiku algustähed a, b, c, d, ... ja lõpu tähed x, y, z kasutuses avaldistes ja muutujateks jäid vaid tähestiku keskmised tähed. Samuti mängis siin suurt rolli asjaolu, et väga tihti olid tsüklimuutujate väärtused täisarvud ehk ***integers***, nii et just ***i*** täht ja sealt edasi osutusid populaarseks.

Loomulikult võid sa tegelikult kasutada tsüklimuutujatena ka enda mõeldud nimesid, kuid austusest teiste vastu (vanade programmeerijate vastu), kui ka nende vastu, kes peaksid sinu koodi lugema ja sellest aru saama, siis soovitan sul kasutada ikkagi üldtunnustatud muutuja nimesid. See vähendab tunduvalt ajakulu koodi lugemisel.

Vastupidi aga, et sa kasutad üldtunnustatud tsüklimuutujaid tavamuutuja nimedes, ei ole lubatud. Selline kood näitab halba stiili!

Seega, meie korrutustabeli kood võiks tegelikult välja näha selline:

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

Tsükkel sõnedega

Kõikides eelmistes tsükli näidetes kasutasime tsüklimuutujat kui loendajat või kui arvu. Isegi siis, kui tsükli listi sisuks kirjutasime ['konn', 'karu', 'kass', 'madu', 'lind'] või siiski mitte, tookord lasime tõepoolest ikkagi ju ka tsüklimuutuja väärtuse välja printida. Seega ehk sa juba aimad, mis selles peatükis jutuks tuleb.

Tõesti, tsükli listi sisu ei pea olema arvude jada, vaid võib olla ka tekstide jada või mis iganes muude objektide jada. Ava uus Idle tekstiredaktori aken, kirjuta järgmine näide ja pane käima!

```
for i in "Tere, maailm!":
    print(i)
```

Põnev, mis? Kirjutasime tavapärase listi asemele teksti ja samuti läks for-tsükkel tööle. Nimelt vaatab Python suvalist sõnet samuti listina, kus iga täht, sümbol, tühik vms on kui üks listi element selles jadas. Seetõttu töötabki for-tsükkel ka suvalise tekstiga, kus iga tsükli tiiru korral saab tsüklimuutuja väärtuseks järgmise sümboli sõnest. Tsükkel töötab nii kaua, kuni kõik elemendid otsa saavad.

Miks ta paneb aga iga tähe ja märgi eraldi reale? See on juba print() käsu omapära, print() käsu sisse on reavahetus sisse programmeeritud. Kui sa tahad, et kõik tähed tuleksid ikkagi ühele reale, siis tuleb print() käsu viimaseks argumendiks kirjutada **end=""**, mis nö kustutab reavahetuse (argumendid eraldatakse alati üksteisest komaga).

Veel üks näide:

```
for i in ['AngryBirds', 'Minecraft','Human Revolution', 'The Sims']:
    print(i, "on lahe mäng")
```

AJAMÕÕTJA!

Aeg jälle mängudes tihti kasutusel olevate osade õppimiseks. Väga sageli tuleb mängudes midagi aja peale teha. Kuidas panna aga arvuti sekundeid lugema, seda me just selles peatükis vaatamegi.

Ajamõõtja võib tööle panna ka ilma nö visuaalse pooleta, kuid õppimise ajal, kuidas ma saan kindel olla, et minu programm ikka õigesti töötab, sellepärast kasutame järgmises koodis ka iga sekundi välja printimist. Jällegi, kirjuta järgmised read enda IDLE tekstiredaktorisse, salvesta ja pane käima.

```
import time

for i in range(10, 0, -1):
    print(i)
    time.sleep(1)

print("Aeg läbi!")
```

Arvuti sekundeid lugema panemiseks tuleb Pythonis importida abimoodul **time**. Ning seejärel tekitada tsükkel, mis loendab **range()** funktsiooni abil numbreid 10-st 1-ni.

Mäletate, range() peatükis kirjutasime kõigepealt arvu, kust tahan loendust alustada ja teisena arvu, kus tahan lõpetada (NB! ühe võrra suuremana). Siin näites aga on lõpetamise arv väiksem kui alustamise oma. Seega peame panema ka kolmanda argumendi, mis ütleb, kuidas 10-st 0-ni liigutakse. Alla loendamise saavutame aga ainult negatiivse arvuga. Kuna soovin lugeda iga sekundit, siis selleks negatiivseks arvuks saab olla vaid -1.

Tsükli kehaks on kaks käsku. Iga tsükli tiiru korral, kirjutatakse tsüklimuutuja väärtus ekraanile ja oodatakse 1 sekund (`time.sleep(1)`), seejärel saab tsüklimuutuja uue väärtuse, oodatake taas 1 sekund jne, kuni tsükli listi kõik elemendid on läbi käidud. Peale tsükli läbimist kirjutatakse "Aeg on läbi".

WHILE tsükkel

Oleme pikalt rääkinud ühest tsükli tüübist for-tsüklist. Selle põhiliseks omaduseks on see, et me alati teame mitu korda tsükkel ennast kordab.

Nüüd aga vaatame sellist tsüklit, kus korduste arv on teadmata. Tsükkel lõpetab oma töö vaid juhul, kui enam ei saa täita ette antud tingimust. Klassikaliseks näiteks on arvu arvamise mäng:

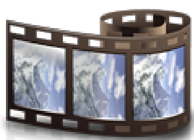
arvaArvu.py - Z:\Teeme ise arvutimängu\minu programmid\arvaArvu.py

File Edit Format Run Options Windows Help

```
import random
arv = random.randint(1,999)
print("Tere! Mis su nimi on?")
nimi = input()
print ("Hei, "+ nimi+ ", arva, millist tuhandest väiksemat arvu ma mõtlen?")
arvamus = int(input())
loendur = 1

#järgmine koodijupp on tsükkel, mis töötab nii kaua, kuni arv erineb
#algselt mõeldud arvust või kuni arvatud on liiga palju

while arvamus != arv :
     # Kas arv on suurem või väiksem
     if arv > arvamus:
         print ("Liiga väike! Paku suuremat arvu!")
     elif arv < arvamus:
         print ("Liiga suur! Paku väiksemat arvu!")
     arvamus = int(input())
     # Suurenda arvamuste loendurit ühe võrra
     loendur = loendur + 1
# Katkesta töö, kui kümnest arvamusest ei piisanud
     if loendur > 10 :
         break

if loendur <= 10 :
     print ("Tubli, "+nimi+ "! Sa arvasid minu arvu", loendur, "korraga." )
else :
     print ("Kümnest arvamisest ei piisanud! Äkki tahad taktikat muuta?")
```

Kuidas WHILE-tsükkel töötab?

- Kirjutatakse võtmesõna **WHILE**, mille taha tuleb lisada **tingimus** ja **koolon**.
- Kooloni järele uuele reale tuleb **taandega** (4-tühikut) kirjutada käsud, mida hakatakse täitma juhul, kui WHILE-tingimus **ON täidetud**.
- Kui kogu WHILE-tsükli keha käsud on täidetud, minnakse uuesti tagasi WHILE-tsükli algusesse ja kontrollitakse uuesti, kas tingimus, mis on seal kirjas, on täidetud?
- Kui **JAH**, siis hakkab tsükkel uuesti otsas peale oma keha täitma, kui **EI**, siis ei täideta peale koolonit enam ühtegi rida ja minnakse tsüklist välja järgmiste käskude juurde programmis.



Jäta tsükkel pooleli!

On olukordi, kus on vaja jätta tsükli töö pooleli. Kas enne seda, kui FOR-tsükkel on kõigi listi elementidega ühele poole saanud või enne seda kui WHILE-tsüklis tingimus enam ei kehtiks.

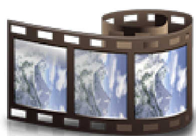
Kui olid tähelepanelik, siis ühte katkestamise meetodit kasutati arvu arvamise mängus ja katkestamise tingimuseks oli seal arvatud kordade arv > 10. Loomulikult ei pidanud sa siiamaani seda veel täielikult mõistma. Nüüd aga selle juurde väike selgitus.

Tsükli tööd võib katkestada kahel erineval viisil:

- jätan tsükli ajutiselt pooleli st jätan näiteks paar tsükli tiiru vahele - seda tehakse käsuga **`continue`**
- lõpetan tsükli töö lõplikult - seda tehakse käsuga **`break`**

CONTINUE vs BREAK

Mõne tsüklitiiru vahelejätmiseks kasutatakse **CONTINUE** käsku. Kui on vaja tsükkel lõplikult jätta pooleli, siis tuleb kasutada **BREAK** käsku. Vaatame nende kahe käsu erinevusi mõlema tsüklitüübi korral videost. Soovitan sul avada kõrval aknas ka oma Idle tekstiredaktor ja näited seal ise järele proovida.



Kommentaarid

Kõikide programmikeste näidetes va arvu arvamise mängus olid meil vaid käsud arvuti jaoks. Kuid väga kasulik on kohe alguses õppida kirjutama koodi sisse kommentaare ka iseenda või teiste jaoks, kes sinu koodi loevad. Vaata näiteks arvu arvamise mängu koodi. Seal on mitmeid punaseid ridu, mis ei tähenda arvuti jaoks mitte midagi, kuid inimese jaoks, kes seda koodi loeb, võivad saada paljud asjad märksa selgemaks ja arusaadavamaks.

Lisaks selgusele on kommentaaridel väga oluline roll ka programmi **dokumenteerimisel**. Mida see tähendab? Dokumentatsioon on midagi kasutusjuhendi taolist ja vastab tüüpiliselt umbes sellistele küsimustele:

- Milleks on antud programm kirjutatud?
- Kes on selle programmi kirjutanud?
- Kellele on see programm kasutamiseks mõeldud?
- Kuidas on programmi töö organiseeritud?
- Ja nii edasi ...

Kuidas lisada kommentaare?

Kommenteerida saab mitmel erineval moel.

Üherealine kommentaar

Iga rea võib muuta kommentaariks, kui panna rea algusesse trellide (**#**) märk. Märgi taga olev tekst muutub IDLE-s punaseks ja on automaatselt sellega Pythonile selgeks teinud, et seda rida ei pea programmi käivitamisel vaatama.

Üherealisi kommenteerimisi kasutatakse ka näiteks mõne päris koodirea välja jätmiseks olukorras, kus programm ei tee päris seda mida ta võiks teha. Nii erinevaid ridu välja kommenteerides on üks võimalus veale jälile jõuda.

Mitmerealine kommentaar

Kui on vaja kirjutada veidi pikemat selgitust mõne programmilõigu kohta, siis tasub kasutada mitmerealist kommenteerimise stiili. Mitmerealise kommentaari saad sa kas iga rea ette trellide panemisega või kasutada **kolmekordseid jutumärke**.

Trellidega mitmerealise kommentaari lisamisel kasutatakse tihti tärnidega kombineeritud varianti, see toob kommentaari väga selgelt muust koodist esile ja on kergesti üles leitav ja loetav. Sellist stiili kasutatakse tihti programmi alguses, kuhu kirjutatakse programmi tegija nimi/nimed, kuupäevad, kellele ja milleks on programm loodud jne.

```
#************
# See on kommentaar!
# See on mõeldud näidismaterjaliks.
# Loodud 6.02.12, Tiina Kull
#************
```

```
"""
Ka see on kommentaar, aga roheline:)
Kõik mis jääb kolmekordsete jutumärkide
vahele, loetakse kommentaariks ja
programmi käivitamisel ei arvestata.
"""
```

Mäng "Arva arvu" uuesti

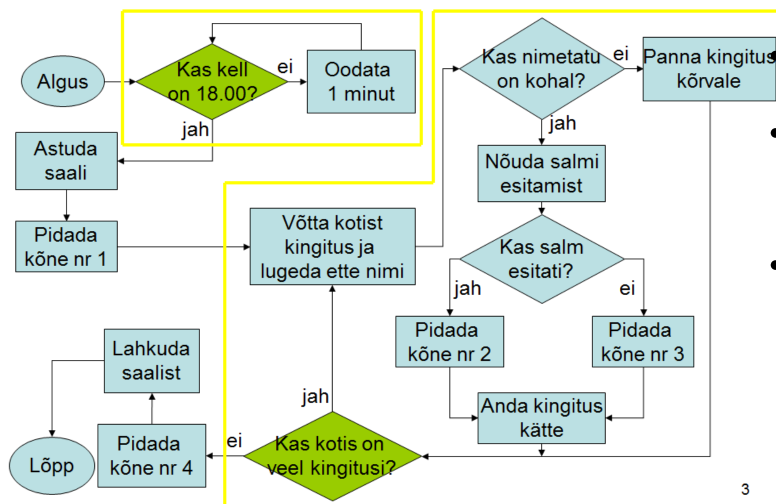
Nii, nüüd tõepoolest oleme üle vaadanud kõik osad selleks, et saada aru Arvu arvamise mängu kõikidest detailidest. Vaatame selle koodi veelkord koos läbi.

Kuidas kavandada mängu?

Enne mängu kirjutama hakkamist tuleb kõigepealt kirja panna ehk kaardistada, mida see mäng tegema hakkab. Alguses üldised punktid, kuni lõpuks päris detailideni välja. Detailideni lahti kirjutamine tähendab tõepoolest samm-sammulist protsessi läbi mõtlemist. Näiteks:

- Kui kasutaja vajutab vasakut nooleklahvi, liigub koer kolm sammu vasakule.
- Kui mängija sisestab arvu, kontrollitakse selle arvu sobivust jne

Programmi algoritmi kirja panemist võib võrrelda maja plaani koostamisega. Tavaliselt ei hakata maja enne ehitama kui pole korralik projekt enne paberil valmis. Nii ka programmidega, kui on vähegi suurem töö, tehakse eelnev kavand. Algoritme pannakse sageli kirja või joonistatakse üles plokkskeemina, siin sulle üks näide jõuluvana tegutsemise algoritmist plokkskeemina:



Plokkskeemi tegemisel tuleb silmas pidada järgmisi reegleid:

- Algoritmil (loe: programmil) on alati **üks algus** ja **üks lõpp** ja neid kujutatakse **ovaalidega**.
- Neutraalsed tegevused nagu näiteks mingi teate andmine, mingi arvu leidmine, loendurite suurendamine, liikumine ühest punktist teise jms pannakse kirja **ristkülikute** sisse.
- **Otsustust nõudavad tegevused** ehk hargnemised pannakse alati **rombi** sisse. Samuti käivad rombi sisse ka tsüklid, kus kontrollitakse mingi tingimuse kehtimist (while). Rombidest väljub seega ALATI kaks noolt, **JAH** ja **EI** suund. Ehk siis olukorrad kui tingimus kehtib ja olukorrad kui ei kehti. Rombi sisse kirjutatakse otsustust nõudev küsimus. Tsüklite mõjualad on joonisel eraldi veel välja toodud kollase piirjoonega. Sellist piiri ei pea tingimata joonistama.
- Kuigi siin pildil selline variant puudub, siis kasutajaga andmete vahetamise tegevused pannakse kirja **rööpküliku** sisse. Näiteks kasutajalt nime küsimine vms.
- Ühtegi lahtist otsa ei tohi plokkskeemis jääda (olukorda, kus plokist ei välju ühtegi noolt) või skeemi jääb lõpetama mitu haru. Alati peab plokkskeemis saama liikuda ALGUSe ovaalist LÕPPovaali.

Mida õppisid?

Vaatame veelkord selle nädala teemad üle. Sa õppisid:

- kuidas panna arvuti otsuseid tegema?
- kuidas töötab If-elif-else konstruktsioon?
- kuidas kontrollida mitut tingimust korraga?
- kuidas kombineerida mitme tingimuse kontrolli ühes if-lauses and ja or ja not abil?
- mis on tsükkel?
- kuidas töötab for-tsükkel?
- mis on list?
- mida teeb range() funktsioon?
- kuidas range() funktsiooni abil arvude vahemikke luua?
- et range() funktsioon algab alati nullist
- et range() funktsioon lõpeb alati ühe võrra enne kui näitab arv sulgude sees
- kuidas range() funktsiooni abil üle mitme elemendi hüpata?
- mida teha siis, kui tsükkel jääb lõputult tööle?
- kuidas programmeerida ajamõõtjat?
- kuidas ja milleks kasutatakse while-tsüklit?
- kuidas tsüklit katkestada lõplikult?
- kuidas jätta mõned tsükli kordused vahele?
- kuidas ja milleks kommentaare programmi lisatakse?
- et kommentaarid on programmeerija enda jaoks (kahe nädala pärast ei mäleta enam, miks just sellise koodi kirjutasin)
- ka programmikoodi võib nö välja kommenteerida vigade leidmise eesmärgil
- kuidas kavandada mängu?
- mis on plokkskeem?

TEE ISE!

Selle nädalaga oled saanud jällegi palju palju targemaks, proovi panna nüüd oma teadmised proovile ja lahenda ka sellel nädalal järgmised 5 ülesannet. Valmis ülesanded tuleb laadida kodutööde esitamise linkide alla moodle keskkonda ja foorumisse. Iga ülesanne eraldi kohta.

1. **Ülesanne: Seiklus_algoritm.** Mõtle ise välja üks lugu, mille võiks realiseerida seiklusmänguna. Umbes midagi sellist nagu oli näide if lause õppimise juures. Ülesanne ei pea olema raskem kui tavaline hargnemine ehk siis tuleks kasutada if-elif-else konstruktsiooni. Siin ülesandes aga tuleb sul teha enda mõeldud mängu kavand ehk siis
    - pane lugu oluliste punktidena kirja (tekstina)
    - joonista üles täpne algoritm plokkskeemina, kuidas mäng algab, kuidas toimub hargnemine, otsustused ja kuidas mäng lõppeb? Võid kasutada graafiku tegemiseks näiteks http://dabbleboard.com/ veebis kasutatavat programmi, kus saad tulemuse pildifailina salvestada. Antud tarkavara ei ole kohustuslik, võib vabalt kasutada ka teisi programme, mis teevad sama tööd.

    Kuidas **dabbleboardi** kasutada:

    - puhasta ekraan (üks nupp vasakul servas)
    - tee hiirega vabakäega kujund - ristkülik, romb, nool vms
    - kirjuta kujundite sisse oluline tekst algoritmi koostamiseks (mine kujundi sisse ja hakka lihtsalt kirjutama)
    - salvesta ja impordi pildifail oma arvutisse, jälgi nuppe tööriistaribal.

    NB! Valmis pildifail lae üles Seiklusmängu_algoritmi esitamise foorumisse, lisades **PILDI**, pildi lisamise ikooni kaudu, **EI** LISA FAILIGA KAASA. Hinda teiste algoritme.

2. **Ülesanne: Seiklus_programm**. Realiseeri oma seiklusmängu kavand Pythonis. Lisa programmi ka paar mõistlikku kommentaari. Pane programmi lõppu input() käsk, sest siis saavad teised kohe ka sinu programmi ilma IDLE-sse laadimata katsetada. Oma programm tuleb üles panna, SEEKORD FAILIGA KAASA!, seiklusmängu esitamise foorumisse. Hinda teiste programme, anna mõistlikke kommentaare.

3. **Ülesanne: Parool.** Kirjuta programm, mis küsib kasutajalt salajast parooli. Programm peab kontrollima, kas sisestatud parool oli õige või mitte. Kui oli õige, siis vastab programm positiivselt (mõtle ise välja midagi, mis siis juhtub), kui mitte, ütleb programm, et parool oli vale ja laseb uuesti parooli sisestada, öeldes midagi sellist, et "küsi parooli selle programmi tegijalt või õpi lugema pythoni koodi". NB! Parooli võib sisestada kokku vaid kolm korda, kui ka siis ei õnnestu tuvastamine, lõpetab programm töö teatega "liiga palju proovisid" vms. (tsükkel, kuni sisestatakse õige parool)

4. **Ülesanne: Ajamõõtja.** Kirjuta ajamõõtja programm ümber nii, et see programm küsiks kasutajalt, mitmendast sekundist aega lugema hakatakse. Programm peab siis ka vastavalt töötama.

5. **Ülesanne: Summa.** Kirjuta programm, mis liidab kõik paaritud arvud 100-st 1000-ni ehk siis arvude 101, 103, 105, ... , 997, 999 summa. Programm peab summa väljastama ekraanile.