Tsükkel

Tükkel on on kolma soluline ehituskivi algoritmide loomisel. **Tsükkel** on keeletarind, mille abil saab osa programmi või algoritmi lauseid korduma panna. Võib ütelda nii, et tsüklilause ise midagi ei tee, kuid ta juhib teiste, tema sees olevate lausete täitmist. Tegelikult käitub sarnaselt ka valikulause. Tsükliga ja tema kasutamisega on seotud mitu olulist küsimust:

- Millal selgub korduste arv (kas on tsüklit alustades see teada või selgub tsükli töötamise käigus)?
- Kes või mis määrab korduste arvu (tingimus)? Kui on enne tskli algust teada, siis kust? Kui selgub tsükli töö käigus, siis mis moodi?
- Milliseid lauseid korratakse ehk mis moodustab tsükli keha ja kuidas seda arvutile selgeks teha?

Tsükleid jaotatakse klassikaliselt kolme tüüpi:

- Eelkontrolliga tsükkel
- Järelkontrolliga tsükkel
- Kindla kordustearvuga tsükkel

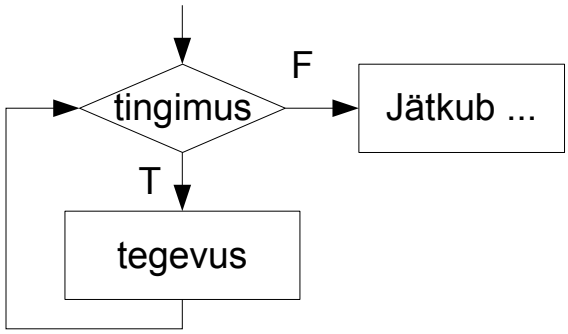
Eelkontrolliga tsükkel

Eelkontrolliga tsükkel on programmeerimiskeeles kõige vajalikum. Selle tsükli olemasolul saab üldiselt kõik „vajadused rahuldada“.

Eelkontrolliga tsükli puhul määrab tsüklis olevate lausete täitmise loogikaavaldis, mis reeglina paikneb tsükli alguses. Kui avaldise väärtus on *true*, siis tsükli sees olevaid lauseid täidetakse / korratakse, kui *false*, siis ei täideta ning kogu algoritmi / programmi täitmine jätkub järgmise lausega peale tsükli lõppu. Et tsükkel üldse nö tööle hakkaks, peab loogikatingimus *true* olema. See võib nõuda vastavate muutujate eelnevat algväärtustmist. Igal juhul tuleb programmi kirjutades jälgida, et ei kasutataks enne muutujat, kui ei ole talle väärtust andnud (ei ole õige panna mingit muutujat juhtima olukorda, kui tema väärtus ei ole programmi poolt eelnevalt määratud). Samas vastavalt olukorrale võib programmi täitmisel juhtuda ka nii, et tsüklis olevad lauseid ei täideta kordagi, sest tingimus oli algusest peale false, kuid ka sel juhul peab olukorra looma muutuja, mis nö teadlikult väärtuse saab..

(Mis on võrdlustehte `a > 10` väärtus, kui ei ole kindlaks määratud a väärtust!?! Kas *True* või *False*?)

Tsükli täitmise käigus peavad loogikatingimuses „osalevate“ muutujate väärtused muutuma. Miks? Sest muidu võib tekkida olukord, kus tsüklit täidetakse igavesti – tingimus jääb *true*'ks ning programm „läheb tsüklisse“.

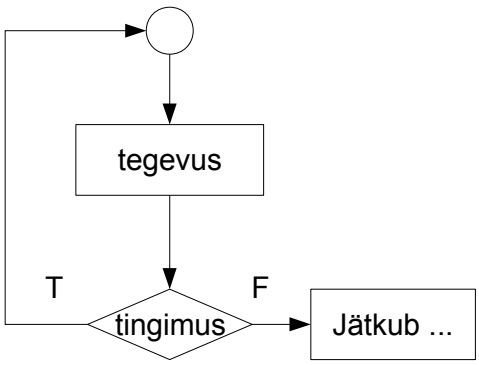


Järelkontrolliga tsükkel

Järelkontrolliga tsükli puhul täidetakse ühe korra tsüklis olevad laused ja seejärel kontrollitakse loogikatingimuse täidetust. Kui tingimus on tõene, täidetakse tsüklis olevaid lauseid uuesti, kui väär, siis tsükli täitmine lõpetatakse. Mõnes keeles võib aga *true/false* olla nö vastupidi, st tsüklit täidetakse siis kui tingimus väär ning tsüklist lahkutakse siis, kui tingimus tõene. Seega oleks siis tegemist tsükli lõpetamise tingimusega. Tõe teadasaamiseks tuleb alati keele reegleid uurida!

Skeem tsükli jaoks on kõrvaloleval joonisel.

Muus osas kasutamistingimused eelkontrolliga tsükliga kattuvad: tsükli täitmise käigus selgub korduste arv, tuleb jälgida, et muutujad oleksid sobivalt väärtustatud. Erinevuseks aga see, et antud tsüklit täidetakse alati vähemalt üks kord.



Kindla kordustearvuga tsükkel

Kui enne tsükli täitma asumist on teada, mitu korda tema sisu täita tuleb, on mugav kasutada kindla kordustearvuga tsüklit. Tavaliselt omab selline tsükkel veel loendurit, mida mitmel erineval otstarbel kasutada saab (lisaks korduste loendamisele). Et selle tsükli toimemehanism keeleti üsna erinev on, siis ei hakka siin skeemi joonistama. Põhimõte aga lühidalt on selline, et enne tsükli alustamist on mingil viisil selgunud korduste arv. Tsüklil on vahendid kordumiste arvu loendamiseks ja arve pidamine selle üle on jäetud tsükli enda kanda. Programmi kirjutaja asi on määrata, mille järgi korduste arv teada saadakse. Reeglina ei ole seda tsüklit ilus kasutada siis, kui korduste arvus tsükli täitmise ajal mingid muudatused tekivad.

Pythoni while-tsükkel

Pythoni eelkontrolliga tsükkel on nn `while`-tsükkel. Keelesõna `while` on sellise tsüklitüübi jaoks üsna tavaline. Tsüklilause üldkuju on järgmine:

```
while loogikaavaldis:
    tegevused
```

Tsükkel toimib üldiselt eelpool kirjeldatud põhimõttel. Sarnaselt if-lausega tuleb kasutada taanet, et näidata lausete kuulumist või mittekuulumist tsükli kehasse, st tuleb eraldada vastav lausete plokk. Erinevuseks on, et loogikaavaldise asemel võib olla ka aritmeetikaavadis ja sel juhul lõpetatakse tsükli täitmine juhul, kui avaldise väärtuseks saab 0. Ehk siis suvaline aritmeetikaavaldise väärtus vastab *True*-le ja 0 vastab *False*-le. Siiski ei soovitata seda võimalust eriti pruukida, sest targem on kasutada nö loetavamaid ja mitte väga eripäraseid konstruktsioone.

Näide: (näidete kaustas nime `while_kordus.py` all:)

```
# muutuja loendur näitab, mitmendat korda tsüklit korratakse ja
# ühtlasi peab arvet korduste arvu ning lõpetamise üle
# NB! seda programmi võiks pigem kindla korduste arvuga tsükliga esitada.
loendur = 1
while loendur < 10:
    print ("Tsüklit täidetakse:", loendur, "korda.")
    loendur = loendur + 1  # Sama tehte teeks ka omistuslause loendur += 1
```

Näide: (näidete kaustas `while_kordus_1.py` nime all)

```
# Järgnev näide kasutab võimalust tsükli täitmist kontrollida
# aritmeetikaavaldisega - loendurit vähendatakse, kuni väärtuseks saab 0.
loendur = 10
```



```
while loendur:
    print ("Tsüklit täidetakse:", loendur, "korda.")
    loendur = loendur - 1  # Sama tehte teeks ka omistuslause loendur -= 1
print ("See oli üks väga hea programm")
```

Loendamine ja summeerimine

Loendamine ja summeerimine kuuluvad väikeste tüüpalgoritmide loetelusse. Kui programmi töö käigus on vaja leida näiteks positiivsete arvude hulk, lugeda üle arvestuse saanud üliõpilased või testis antud õiged vastused tuleb kasutada loendamist. Loendamise üldpõhimõte on järgmine: võetakse üks muutuja, mille abil loendatakse (vali talle sobiv nimi). Loendur on reeglina täisarvu tüüpi. Enne loendamise algust algväärtustatakse loendur nulliga `(loendur = 0)`. Järgneva tegevuse käigus (tavaliselt toimub tegevus tsüklis) suurendatakse loendurit iga kord, kui loendamist vajav sündmus juhus või sobiv väärtus leiti (vt 1. näidet, seal loendatakse tsükli kordumiste arvu). Loenduri suurendamine toimub klassikaliselt lausega

`loendur = loendur + 1`

(loenduri hetkel kehtivale väärtsele liidetakse 1 ja tulemus pannakse samasse muutujasse tagasi).

Erinevad keeled pakuvad sedalaadi tehteks ka erivõimalusi. Pythonis on olemas tehtemärk +=, mida inglisekeeli kutsutakse *augmented assignment* – **täiendatud omistamine**. Lause loenduri suurendamiseks näeb vastavalt välja:

```
loendur += 1
```

Summeerimine on sarnane tegevus, kui 1 asemel liidetakse juurde summasse minevaid arve ja peale iga liitmistehet kasvab summa lisatud arvu võrra:

```
summa = summa + arv
```

ehk

```
summa += arv
```

Summeerimine on siin kasutusel tinglikus tähenduses, sest samahästi võib ka lahutada, korrutada, jagada ja astendada. Ka selleks on Pythonil olemas oma spetsiaalsed täiendatud omistamise märgid, mida võib kasutada lisaks universaalsele vormile – kogu tehte väljakirjutamisele. Need tehtemärgid on:

```
+=   -=   *=   /=   %=   **=
```

Tähendused tulenevad vastavate tehtemärkide tavakasutusest.

Pythoni for-tsükkel

Nagu juba eespool mainitud, töötavad kindla kordustearvuga tsüklid keeleti erinevalt. Tihti kannavad need tsüklid nn **for-tsükli** nime. Pythonil on olemas for-tsükkel, kuid tema käitumine ei ole seda tüüpi tsükli kohta päris klassikaline. Järgnev ülevaade saab edaspidi täiendust, kui teadmistesse lihtandmetüüpide kõrvale lisanduvad struktuursed andmetüübid. Esialgu for-tsükli lihtsamast variandist. For-tsükli üldkuju on järgmine:

```
for tsüklimuutuja in hulk:
    korratavad laused
```

Siin „hulk“ on kõige nö kirevam osa kogu lauses ja nõuab teadmisi struktuursetest andmetüüpidest (mis varsti ka tuleb).

Järgmine näide demonstreerib for-tsükli kasutamist tavalise kindla kordustearvuga tsüklina, kus korduste arv antakse ette täisarvuna (NB! Seda ei pea tegema tingimata programmi kasutaja):

```
kordusi = int(input("Mitu kordust? "))
for i in range(kordusi):
    print (i)
```

Funktsioon range()

Funktsiooni **`range()`** ülesandeks on väljastada/tagastada täisarvude **loend** (*list*). Funktsiooni täielik kirjeldus:

```
range([start,]stop[,step]) -> list of integer
```

Et seda loendit saab edukalt (tuleb tihti) kasutada for-tsükli juhtimiseks, siis funktsioonist veidi põhjalikumalt.

Kui parameetrina on määratud üks täisarv (*stop*), tähistab see täisarvude jada lõppu, kus juures jada algab 0-ga:

```
range(5) -> 0, 1, 2, 3, 4
```

Kui on määratud algus ja lõpp (*start* ja *stop*), siis väljastatakse täisarvud alguse (kaasaarvatud) ja lõpu (väljaarvatud) vahel:

```
range(1,5) -> 1, 2, 3, 4
```

Kolmas parameeter annab sammu (step), mille võrra täisarvu suurendatakse järgmise arvu saamiseks (vaikimisi on samm 1):

```
range(1,10,2) -> 1, 3, 5, 7, 9
```

Loendisse satub 0 täisarvu, kui algus ja lõpp on võrdsed või lõpp on algusest väiskem (positiivse sammu korral). Samm võib olla ka negatiivne.

Nagu parameetrite puhul ikka, võib need etteanda muutujatena. Seega võib täisarvude loend sõltuda eelnevast programmi käitumisest, kasutaja soovist vms. Ning `range()`-funktsiooni abil saab erinevatest tingimustest sõltuma panna ka for-tsüklit, tema korduste arvu.

Näide: kasutaja otsustab, mitut arvu ta ruutu tõsta soovib, seejärel laseb programm tal täpselt nii palju arve sisestada:

```
print ("Mitut arvu soovid ruutu tõsta? ")
mitu = int(input("> "))
print ("Aitäh!\nSisesta arve, kohe näed ka tulemust.")
for i in range(mitu):
    arv = int(input(str(i+1) + ". arv> "))
    ruut = arv ** 2
    print ("Ruut on ",ruut)
```

Edaspidi for-tsükli võimaluste tutvustamine jätkub.

Järekontrolliga tsüklit Pythonis ei ole.

Pythoni break-lause

Tavaliselt on keeltes olemas lause(d), mille abil saab tsükli täitmise suvalisel hetkel pooleli jätta ja tsüklist välja tulla. Programmi täitmine jätkub tsüklile järgneva lausega. Kui tegemist on sisemise tsükliga, tullakse välja vaid sisemisest tsüklist. Pythonis (ja ka teistes keeltes) on lause nimeks **`break`**. Lause kasutamine peaks olema sõltuv mingitest tingimustest (st ta peaks olema seotud if-lausega). Programmeerimise heast stiilist lähtudes on parem, kui breaki liiga kergekäeliselt ei kasutata. Tegemist peaks olema ennekõike hädaolukorraga. Samas võib ta teinekord anda tulemuseks kergemini loetava koodi.

