Tsükkel

Tükkel on kolmas oluline ehituskivi (jada ja tingimuslause kõrval) algoritmide loomisel. Terministandardi sõnastiku (http://eki.ee/dict/its) järgi on eestikeelseks mõisteks tegelikult **silmus** *(ingl loop).* Silmus on lausete või käskude jada, mida teatud tingimuste täidetuse korral saab täita iteratiivselt.

Võib ka öelda nii, et tsükkel on keeletarind, mille abil pannakse osa programmi või algoritmi lauseid korduma. Tsüklilause ise midagi ei tee, kuid ta juhib teiste, tema sees olevate lausete täitmist.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 tsükli 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?

Tsüklite tüübid

Tsükleid jaotatakse klassikaliselt kolme tüüpi (kasutan järgnevalt taas Terministandardi sõnastikku):

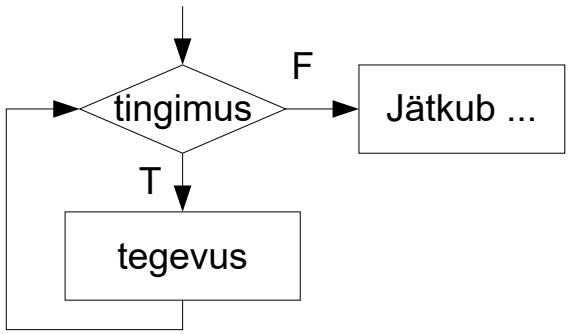
- **eestestiga silmus** *(ingl pretest loop)* - silmusejuhik, mis sooritab testi enne sisenemist silmuse kehasse. Ka eelkontrolliga tsükkel. **Silmusejuhik** *(ingl loop control)* on keeletarind, mis sisaldab testi otsustamiseks, kas silmuse iteratsioon tuleb täita.
- **tagatestiga silmus** *(ingl posttest loop)* - silmusejuhik, mis sooritab testi pärast silmuse keha. Ka järelkontrolliga tsükkel
- **kindla kordustearvuga tsükkel**

Eelkontrolliga tsükkel ehk eestestiga silmus

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

Tsüklis olevate lausete täitmise määrab tsükli alguses paiknev loogikaavaldis (silmuse tingimus ingl *loop assertion*). Kui tsüklitingimuse väärtus on tõene (ingl *true*), siis tsükli sees olevaid lauseid täidetakse / korratakse, kui väär (ingl *false*), siis ei täideta ning kogu algoritmi / programmi täitmine jätkub järgmise lausega peale tsükli lõppu. Et tsükkel üldse nö tööle hakkaks, peab tsüklitingimus tõene 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 programmis eelnevalt määratud). Samas vastavalt olukorrale võib programmi täitmisel juhtuda ka nii, et tsüklis olevad lauseid ei täideta kordagi, sest tingimus oli algusest peale väär.

Tsükli täitmise käigus peavad tsüklitingimuses osalevate muutujate väärtused muutuma. Miks? Sest muidu võib tekkida olukord, kus tsüklit täidetakse igavesti: tsüklitingimus ei muutu vääraks



Tsükli täitmine (vt ka joonist):

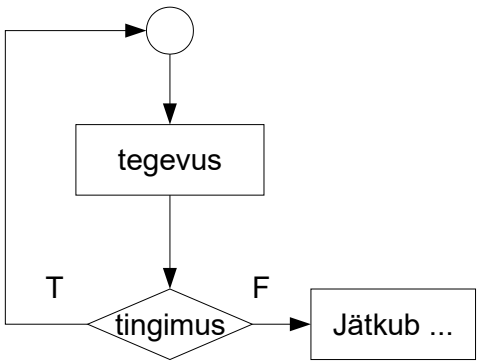
1. Arvutatakse tsüklitingimuse väärtus
2. Kui väärtus on `true`, täidetakse tsükli sees olevad laused ja täitmine antakse tsükli algusesse tingimuse väärtuse leidmiseks.
3. Kui väärtus on **`false`**, lõppeb tsükli täitmine ja programmi jätkatakse käsuga peale tsüklit.

Järelkontrolliga tsükkel ehk tagatestiga silmus

Järelkontrolliga tsüklis täidetakse ühe korra tsüklis olevad laused ja seejärel kontrollitakse tsüklitingimuse 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 kohta 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 korduste arvu loendamiseks ja arve pidamine selle üle on jäetud tsükli enda kanda. Programmeerija ülesanne 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. Sõltuvalt keelest võib seda liiki tsüklit käsitleda ka kui eelkontrolliga tsüklit.

Pythoni while-tsükkel

Keelesõna **`while`** on sellise tsüklitüübi jaoks üsna tavaline ja seda kasutab ka Python. 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 (ingl *loop body*), st tuleb eraldada vastav lausete plokk. Loogikaavaldise asemel võib olla ka aritmeetikaavaldis ning tsükli täitmine lõpetatakse, kui tsüklitingimuse väärtuseks saab 0. Sest väärtus 0 käsitletakse kui **`false`** ning muud suvalist muud väärtust kui **`true`**. Siiski ei soovitata seda võimalust eriti pruukida, sest targem on kasutada inimloetavamaid ja mitte väga eripäraseid konstruktsioone.

Näide 1:

```
loendur = 0
while loendur < 10:
    loendur = loendur + 1
    print ("Tsüklit täidetakse:", loendur, "korda.")
```

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 muutuja, mille abil loendatakse (vali talle sobiv nimi). Loendur on reeglina täisarv. 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 liitomistus (ingl *augmented assignment / compound assignment*) tehtemärk `+=` Lause: `loendur += 1`

Summeerimine on sarnane tegevus, kuid 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 liitomistuse märgid, mida võib kasutada lisaks universaalsele vormile – kogu tehte väljakirjutamisele.

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

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 tsüklimuutuja in hulk:
    korratavad laused
```

Siin „hulk“ on kõige kirevama tähendusega osa kogu lauses ja nõuab teadmisi struktuursetest andmetüüpidest.

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 **järjend** ehk **list** (ingl *list*). Funktsiooni täielik kirjeldus:

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

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

Kui parameetrina on määratud üks täisarv (stop), tähistab see täisarvude jada lõppu, kusjuures 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
```

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

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

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.

Tsüklite kasutamine

Probleemid ja tähelepanekud tsüklite kavandamiseks:

1. Millist tsüklit kahest võimalikust on otstarbekas kasutada? Kui korduste arv on enne tsüki algust teada ja mitte miski tsükli sees toimuv seda segada ei saa, tasub kasutada for-tsüklit. Lihtsam on kirja panna kordamise tingimusi. Lisaväärtust omab tsüklimuutuja massiivide töötlemisel indeksi(te) genereerimiseks või listi elementidde ükshaaval kättesaamiseks. Ülejäänud juhtudel on pigem sobiv while-tsükkel.
2. Leia üles laused, mis peavad tsükli sisse kuuluma. Ehkki see tundub triviaalsena, on õige lausetekomplekti tuvastamine sageli (eriti alguses) probleeme tekitav.
3. Koosta sobiv tsüklitingimus. Tihti ei aita ühest võrdlustehtest, vaid on vaja kontrollida erinevate muutujate väärtusi korraga. Sel juhul võib vigu tekitada loogikatehete **`and`** ja **`or`** kasutamine. Samal ajal tuleks ka jälgida, et loogikatingimuse väärtus tsükli sees üldse muutuda saab. Kui kasutada kontrollimisel muutujat, millega tsükli sees midagi ei tehta, on ilmselt kusagil viga. Vigane ei pruugi kindlasti olla tingimus, vaid tsükli kehas võivad sel juhul puududa mõned olulised laused.
4. Õigete võrdlustehete kasutamine loogikatingimuses võib olla määrava tähtsusega tsükli korrektsuse tagamisel, õige korduste arvu saavutamisel, aga ka lõputu tsükli vältimisel. Sagendane viga on ühe võrra suurem või väiksem korduste arv.
5. Enne tsükli alustamist (eriti while-tsükli puhul) tuleb jälgida ka loogikatingimuses osalevate muutujate algväärtustamist, sest nende muutujate väärtusi asutakse kohe kontrollima. Kui õiged väärtused on jäänud kogemata omistamata, võib tsükkel mitte käivituda.

