TARTU ÜLIKOOL
Arvutiteaduse instituut
Infotehnoloogia mitteinformaatikutele õppekava

Heidi Meier

Õppijate käitumismustrid programmeerimisülesande lahendamisel: logifailide analüüs

Magistritöö (15 EAP)

Juhendaja: PhD Eno Tõnisson

Tartu 2018

Õppijate käitumismustrid programmeerimisülesande lahendamisel: logifailide analüüs

Lühikokkuvõte:
Programmeerimise õppimisel on suur osa iseseisval ülesannete lahendamisel ning õpetajad ei näe, kuidas õppijad tulemuseni jõuavad, eriti on see nii MOOC-idel (vaba juurdepääsuga e-kursustel). Programmeerimisülesannete lahendamise protsessi uurimine võib anda õpetajale väärtuslikku informatsiooni. Magistritöös seati eesmärgiks programmeerimiskeskkonna Thonny logifailide põhjal leida vastused järgmistele küsimustele: Millist informatsiooni on võimalik Thonny logifailidest koguda õppijate tegevuse kohta? Millised on õppijate erinevad käitumismustrid programmeerimisülesannete lahendamisel MOOC-i "Programmeerimise alused" kokkuvõtva arvestusülesande näitel? Milliseid õppijatüüpe saab eristada ülesande lahendamise käitumismustrite alusel? Õppijate käitumismustrite analüüsimisel võeti aluseks, kas nad lahendasid ülesannet järk-järgult või mitte, s.t kas nad kirjutasid programmi osade kaupa ja vahepeal ka testisid selle tööd või testisid esimest korda alles siis, kui olid suurema osa lahendusest valmis kirjutanud. MOOC-i kokkuvõtva arvestusülesande lahenduste kontekstis saab öelda, et nendest seitsmest õppijast, kellel läks kõige kauem aega ning oli kõige rohkem veateateid ja korduvaid veateateid, ei töötanud keegi järk-järgult.

Võtmesõnad:
Õppijatüübid, Pythoni veateade, logifail, Thonny, programmeerimise algõpe, MOOC

CERCS: P175 Informaatika, süsteemiteooria, S281 Arvuti õpiprogrammide kasutamise metoodika ja pedagoogika

Behaviour Patterns of Learners at Solving Programming Task: an Analysis of Log Files

Abstract:
Solving individual tasks plays an important role while studying programming and teachers do not see how learners achieve the result, especially in case of MOOCs (massive open online courses). Investigating the process of solving programming tasks can provide the teacher with valuable information. The aim of this thesis is to find answers to the following questions based on the log files of the programming environment Thonny: What information can be collected from Thonny's log files about learners' behaviour? What are different learners' behaviour patterns of solving programming tasks based on the example of a summary task of a MOOC "Introduction to Programming"? What types of learners can be distinguished on the basis of the behaviour patterns of task solving? The analysis of the learners' behaviour patterns was based on whether they solved the task gradually or not, i.e. whether they wrote the programme in parts, and in the meantime also tested their work or tested it for the first time only when they had written the bulk of the solution. In the context of the solution of the MOOC's summary task, one can say that the seven learners who spent the most time and who had the greatest number of error messages and repetitive error messages, did not work gradually.

Keywords:
Student profiling, Python error message, log file, Thonny, introductory programming, MOOC

CERCS: P175 Informatics, systems theory, S281 Computer-assisted education

Sisukord

Sissejuhatus

Üha rohkem erinevas eas inimesi õpib programmeerimist ja õpetamise paremaks korraldamiseks on vaja täpsemalt teada saada, kuidas nad õpivad. Programmeerimise õppimisel on suur osa iseseisval ülesannete lahendamisel ning õpetajad ei näe, kuidas õppijad tulemuseni jõuavad, eriti on see nii MOOC-idel (vaba juurdepääsuga e-kursustel). Samas võiks just õppijate tegevuste uurimine anda väärtuslikku informatsiooni selle kohta, kuidas ja kui erinevalt õppijad ülesandeid lahendavad. Teadmised õppeprotsessi kohta annavad võimaluse täiustada õppemetoodikat, luua veel paremaid ülesandeid ja abimaterjale.

Tartu Ülikooli õppejõu Aivar Annamaa väljatöötatud Pythoni programmeerimiskeskkond Thonny talletab programmeerimise käigus tehtavate toimingute info logifaili. Seega on logifailide näol olemas väärtuslikud algandmed, mida on võimalik eri meetoditega analüüsida ja nii saada õppeprotsessi kohta teada mitmekülgset informatsiooni.

Thonny logifailide põhjal on kirjutatud magistritöö (Kodasmaa, 2017a), mis analüüsib ja kirjeldab veateadete tüüpe. Analüüsile lisaks on tema töös koostatud programmeerimise e-kursuste jaoks abimaterjalid, mis aitavad õppijatel veateadetest paremini aru saada. Samuti on logifaile analüüsitud bakalaureusetöös (Pedel, 2016), mille rõhuasetus on failidest info lugemisel ja selle kirjeldamisel, et õppejõududel ja järgmistel uurijatel oleks olemas abistav tugimaterjal. Lisaks on koostatud logifailide analüüsiprogramme (Aramaa, 2014).

Ka mujal maailmas on suund programmeerimise õppimise protsessi detailsema uurimise poole, kusjuures uurimine on hoogustunud viimase 15 aasta jooksul. Kõige enam on uurimusi tehtud Java põhjal. Paljud uurimused on võtnud aluseks kompileerimised ning nende põhjal teinud järeldusi õppeprotsessi kohta (Jadud (2005, 2006b); Allevato ja Edwards (2010); Altadmri (2015); Denny (2012); Watson ja Godwin (2013)). Viimastel aastatel on hakatud üha enam tähelepanu pöörama protsessi detailsemale jälgimisele, kasutades logimise (näiteks kõigi klahvivajutuste registreerimise) funktsionaalsust. Detailsemat uurimist kasutavad nt Vihavainen, Helminen ja Ihantola (2014) (Java), Marceau, Fisler ja Krishnamurthi (2011a, 2011b) (Racket) ning Blikstein (2011, 2013) (NetLogo). Pythoni veateateid on uurinud Pritchard (2015), et selgitada, millised neist vajavad algõppes lisaselgitusi. Õppijaid on tüüpidesse jaganud näiteks Blikstein (2011) ja Hosseini et al. (2014).

Õppijate tegevus programmeerimisülesande lahendamisel võib mitmel moel erineda. Näiteks lahendab osa ülesannet järkjärguliselt, seejuures igas etapis ka testides, teised kirjutavad kohe

kogu programmi peaaegu valmis. Osa kasutab koodiosade väljakommenteerimist, teised mitte. Osa kasutab varasemate ülesannete lahendusi, teised ei kasuta. Osa käivitab mõnikord pärast veateadet programmi uuesti, kui ei ole teinud koodis muudatusi, teised mitte. Sellelaadsete erinevuste põhjal saab rääkida erinevatest käitumismustritest, mis ongi selle magistritöö fookuses.

Selle magistritöö eesmärk on logifailide põhjal leida vastused järgmistele küsimustele:

1) Millist informatsiooni on võimalik Thonny logifailidest koguda õppijate tegevuse kohta?
2) Millised on õppijate erinevad käitumismustrid programmeerimisülesannete lahendamisel MOOC-i "Programmeerimise alused" kokkuvõtva arvestusülesande näitel?
3) Milliseid õppijatüüpe saab eristada ülesande lahendamise käitumismustrite alusel?

Esimene peatükk annab magistritööle teoreetilise raamistuse, käsitledes uuringu taustaga seotud olulisi aspekte. Vaatluse all on programmeerimise algõpe, Pythoni veateated ja klassifikatsioon ning programmeerimiskeskkond Thonny, kusjuures fookuses on logifailide funktsionaalsus. Selle peatüki kõige olulisem osa on lühiülevaade varasematest uuringutest, mis käsitlevad õppimist programmeerimise algõppes. Teine peatükk kirjeldab uurimuse materjali ja metoodikat. Kolmas peatükk keskendub õppijate käitumismustrite analüüsile programmeerimisülesande lahendamisel. Eraldi on tähelepanu all logifailides sisalduv informatsioon ja selle kirjeldamine ning analüüs.

Analüüsi aluseks on 2017. aasta sügis-talvel toimunud Tartu Ülikooli MOOC-i “Programmeerimise alused” kokkuvõtlik arvestusülesanne (ülesande tekst on alapeatükis 3.2.1). Kasutati ainult neid logisid, mis asusid ühes failis ning samas failis ei olnud teiste ülesannete lahenduskäike.

1. Programmeerimise algõpe, veateated ja logid

See peatükk annab teoreetilise raamistuse programmeerimise ülesannete lahendusprotsessi käsitlevale uurimusele. Esimene alapeatükk käsitleb programmeerimise algkursusi, keskendudes uurimuse andmete aluseks olevatele Tartu Ülikooli e-kursustele, mis järgivad MOOC-i põhimõtteid. Teine alapeatükk käsitleb veateateid, millele reageerimine on programmeerimise õppimise protsessi oluline osa. Lisaks vigade klassifikatsioonile on tähelepanu all ka erinevate veatüüpide sagedused. Kolmas alapeatükk tutvustab programeerimiskeskkonda Thonny, võttes fookusesse logifailide funktsionaalsuse, mis on käesoleva uurimuse aluseks. Neljas alapeatükk keskendub õppimist käsitlevatele uurimustele programmeerimise algõppes.

1.1 Programmeerimise algkursused

Programmeerimise algkursuste maastik on väga mitmekesine: algtõdesid on võimalik õppida koolides, samuti korraldatakse erinevaid koolitusi ning õppimisvõimalusi on ka rahvaülikoolides. Osa kursusi kasutab olulise osana kontakttunde: toimuvad loengud ja praktikumid, millele lisanduvad kodused ülesanded. Lisaks on väljatöötatud kursusi, kus peamine roll on e-õppel. Just e-kursuste puhul on õppeprotsessi uurimine logifailide põhjal eriti oluline, kuna õpetaja ei näe, kuidas õppijad lahenduseni jõudsid ja millised raskused neil tekkisid.

Kuna magistritöö kasutab analüüsiks Tartu Ülikooli e-kursuse "Programmeerimise alused" ülesannet (ülesande tekst on peatükis 3.2.1) ja osalejate lahenduskäikude andmeid, siis käsitletakse järgnevalt põgusalt ka MOOC-e, mille põhimõtteid nimetatud kursus järgib, samuti teisi samalaadseid Tartu Ülikooli programmeerimise e-kursusi.

MOOC on akronüüm sõnadest *massive, open, online, course* (rohkearvuline avatud veebipõhine kursus). Mõiste võeti esmakordselt kasutusele 2008. aastal kirjeldamaks Manitoba ülikooli loodud avatud registreerumisega veebipõhist kursust *Connectivism and Connective Knowledge,* millest võttis osa enam kui 2200 õppijat (Mackness, Mak & Williams, 2010).

2015. aastal anti välja raport “Institutional MOOC strategies in Europe”, kus määratletakse ka MOOC-i definitsioon ja sisu. Väljatöötatud määratluse järgi on MOOC-id tasuta ja kõigile avatud veebipõhised kursused, mis on koostatud suurt hulka osalejaid silmas pidades (Jansen & Schuwer, 2015).

Võib öelda, et e-kursus peab vastama järgmistele tingimustele, et olla MOOC: kursus toimub veebipõhises õpikeskkonnas; teadmised on kättesaadavad kõikidele soovijatele (osalejad ei pea olema seotud ühegi õppeasutusega); kursusele registreerumiseks ei ole piiranguid; kursuse katkestamisel ei ole tagajärgi (MOOC – miks, kellele ja kuidas, 2017).

Tartu Ülikooli arvutiteaduse instituut on MOOC-e väljatöötanud alates 2014. aasta detsembrist (Lepp et al., 2017). Loodud on programmeerimise algtõdesid tutvustavad kursused "Programmeerimisest maalähedaselt", "Programmeerimise alused" (sh ka õpilaste versioon) ja "Programmeerimise alused II".

"Programmeerimisest maalähedaselt" toimus esmakordselt aastal 2015 ja oli esimene programmeerimist tutvustav MOOC Tartu Ülikoolis ning ühtlasi kõige esimene eestikeelne MOOC (Lepp et al., 2017). Kursus on suunatud erinevas vanuses inimestele, kellel puudub eelnev kokkupuude programmeerimisega. Õpe kestab neli nädalat ja maht on 1 EAP-d ehk 26 tundi (Programmeerimisest maalähedaselt, 2017).

"Programmeerimisest maalähedaselt" on Eestis seni kõige suurema osalejate ja lõpetajate arvuga MOOC, mis valiti ka 2016. aasta parimaks e-kursuseks. Kursuse lõpetajate protsent on ligikaudu 65% registreerunutest, mis on MOOC-ide kontekstis erakordselt suur osakaal (Lepp et al., 2017). Ülemaailmselt on MOOC-ide lõpetajate protsent keskmiselt üsna väike. Kuigi erinevates töödes esitatud arvud on mõneti erinevad, on suurusjärk sama – räägitud on näiteks 6,5%-st (MOOC – miks, kellele ja kuidas, 2017) või märgitud, et keskmine lõpetajate osakaal jääb alla 13% (Onah, Sinclair & Boyatt, 2014) või suurusjärku 5–10% (Jordan, 2014).

Tõsisematele huvilistele mõeldud kursus "Programmeerimise alused" on eelmisest oluliselt mahukam (õpe kestab kaheksa nädalat ja maht on 3 EAP-d ja 78 tundi) ning toimus esimest korda aastal 2016. Selle MOOC-i eesmärk on tutvustada algoritmilist mõtteviisi ja programmeerimist ning sellega seonduvat neile, kel varasem kokkupuude programmeerimisega puudub või on vähene (Lepp et al., 2017; Programmeerimise alused, 2016). Alates 2017. aastast on õppijatel võimalus ennast proovile panna ka samamahulisel jätkukursusel "Programmeerimise alused II".

Nimetatud programmeerimise MOOC-ide materjalide ja ülesannete kasutus on laiem, nimelt kasutatakse neid ka ülikooli statsionaarsetel kursustel, kus osaleb üliõpilasi erinevatelt erialadelt. Selles magistritöös on kasutatud andmeid MOOC-ilt "Programmeerimise alused".

1.2 Pythoni veateated ja klassifikatsioon

Kui programmeerida, tekib protsessi käigus tahes-tahtmata vigu ning on vaja teada saada, mis on läinud valesti. Kasutajale kuvatakse veateade, mis annab informatsiooni vea kohta. Pythoni puhul võib eristada kahte peamist tüüpi veateateid: süntaksivead (ingl *syntax errors*) ja erindid (ingl *exceptions*) (Python 3.6.4 documentation, s.a.).

Süntaksiviga on viga, mille puhul parsimisel (ingl *parsing*) fikseeritakse, et süntaks ei vasta reeglitele, ja programmi ei käivitata. Süntaksivigu nimetatakse aeg-ajalt ka parsimise vigadeks (ingl *parsing errors*) (Python 3.6.4 documentation, s.a.; Schliep, 2015).

Erindi puhul fikseeritakse viga programmi täitmisel (ingl *execution time*), s.t ebaõnnestub mõne konkreetse käsu täitmine. Aeg-ajalt nimetatakse neid ka täitmisaegseteks vigadeks (ingl *runtime error*) (Annamaa, s.a.; Schliep, 2015). Erindeid on mitut tüüpi ning tüübinimi sisaldub veateates, näiteks *NameError* (ee nimeviga), *TypeError* (ee tüübiviga) ja *ValueError* (ee väärtuse viga). Pythoni ametlikus dokumentatsioonis on esitatud ka standard-erindite (ingl *built-in exceptions*) täielik loetelu (Python 3.6.4 documentation, s.a.).

Järgnevalt valik levinumaid veatüüpe, mida käsitletakse enamikes algajatele mõeldud materjalides (Python Errors, 2012; Python 3.6.4 documentation, s.a.).

Süntaksiviga *(*ingl *SyntaxError)* – programmis on õigekirjaviga.
Nimeviga (ingl *NameError)* – muutuja või funktsioon on defineerimata, s.t kasutatakse muutujat või funktsiooni, mida ei ole loodud või mis asub vales kohas.
Tüübiviga (ingl *TypeError)* – püütakse teha toimingut, mida rakendatav andmetüüp ei toeta.
Väärtuse viga (ingl *ValueError)* – argumendiks on muutuja, mille tüüp või väärtus ei sobi (nt ruutjuur negatiivsest arvust).
Indeksiviga (ingl *IndexError)* – järjendis puudub sellise indeksiga element, mida küsitakse, s.t indeks on järjendi piiridest väljas.

Vigade käsitlemisel ei saa tähelepanuta jätta ka loogikavigu (ingl *logic error),* mida nimetatakse paljudes materjalides ka semantikavigadeks (ingl *semantic error*). Selle veatüübi puhul programm tavaliselt kompileerub ja veateadet ei tule, kuid see ei tööta nii nagu peaks. Loogikavigu on sageli ka väga raske avastada (Downey, 2012; Magee, 2014).

Süntaksivead ja erindid esinevad küllaltki erineva sagedusega ning siinkohal on oluline ka programmi keerukus. Downey (2012) märgib, et lihtsate programmide puhul domineerivad

süntaksivead ja erindeid on suhteliselt harva. Tema tähelepanekut kinnitab ka Kodasmaa (2017a) uuring, mis vaatles veateadete sagedusi algajatele mõeldud e-kursusel "Programmeerimisest maalähedaselt" ning jätkukursusel "Programmeerimise alused" (vt kursuste tutvustust ptk 1.1). Nimelt on päris algajatele mõeldud kursuse puhul süntaksivigade osakaal oluliselt suurem – 62% kõikidest veateadetest, jätkukursusel oli vastav osakaal 43%. Erinditest esines kursusel "Programmeerimisest maalähedaselt" kõige enam tüübiviga (24%) ja nimeviga (10%), kursusel "Programmeerimise alused" tüübiviga 30% ja nimeviga 17% kõikidest veateadetest. Teisi erindeid oli mõlema kursuse puhul vähem kui 5%. Kodasmaa (2017a) magistritöö raames valmis ka materjal (Kodasmaa, 2017b), mis käsitleb veatüüpe ja võimalikke lahendusi vigade parandamiseks.

Helminen, Ihantola ja Karavirta (2013) näitasid, kuivõrd on veatüüpide sagedus (mitte vigade sagedus) on seotud konkreetse ülesandega (joonis 1). Seepärast on siinkohal olulised suurte andmehulkade põhjal tehtud uurimused. Näiteks on sageduste analüüsimiseks suuremat veateadete hulka kasutanud Pritchard (2015).

Error	Assign. 1	Assign. 2	Assign. 3
SyntaxError	394 (35%)	195 (17%)	202 (23%)
NameError	274 (24%)	484 (42%)	218 (25%)
IndentationError	173 (15%)	66 (6%)	62 (7%)
TypeError	141 (12%)	252 (22%)	223 (25%)
AttributeError	66 (6%)	135 (12%)	118 (13%)
IndexError	40 (4%)	1 (0%)	25 (3%)
UnboundLocalError	26 (2%)	1 (0%)	15 (2%)
ValueError	21 (2%)	9 (1%)	0 (0%)
ImportError	0 (0%)	15 (1%)	3 (0%)
Other	1 (0%)	7 (1%)	12 (1%)

Joonis 1. Veatüüpide sagedused (Helminen, Ihantola & Karavirta, 2013)

Pritchard (2015) vaatles oma uuringus veateateid ja nende sagedusi, arvestades ka veateates sisalduvat lisainformatsiooni. Nimelt sisaldab veateade lisaks tüübile ka täpsustavat infot, mis võimaldab paremini aru saada, milles tekkinud viga seisneb. Ta analüüsis Pythoni algõppeks mõeldud keskkonna "Computer Science Circles" veateateid. Keskkonnas oli 100 ülesannet ja uuring tehti 1,6 miljoni esituse baasil, millest 640 000 andis veateate.

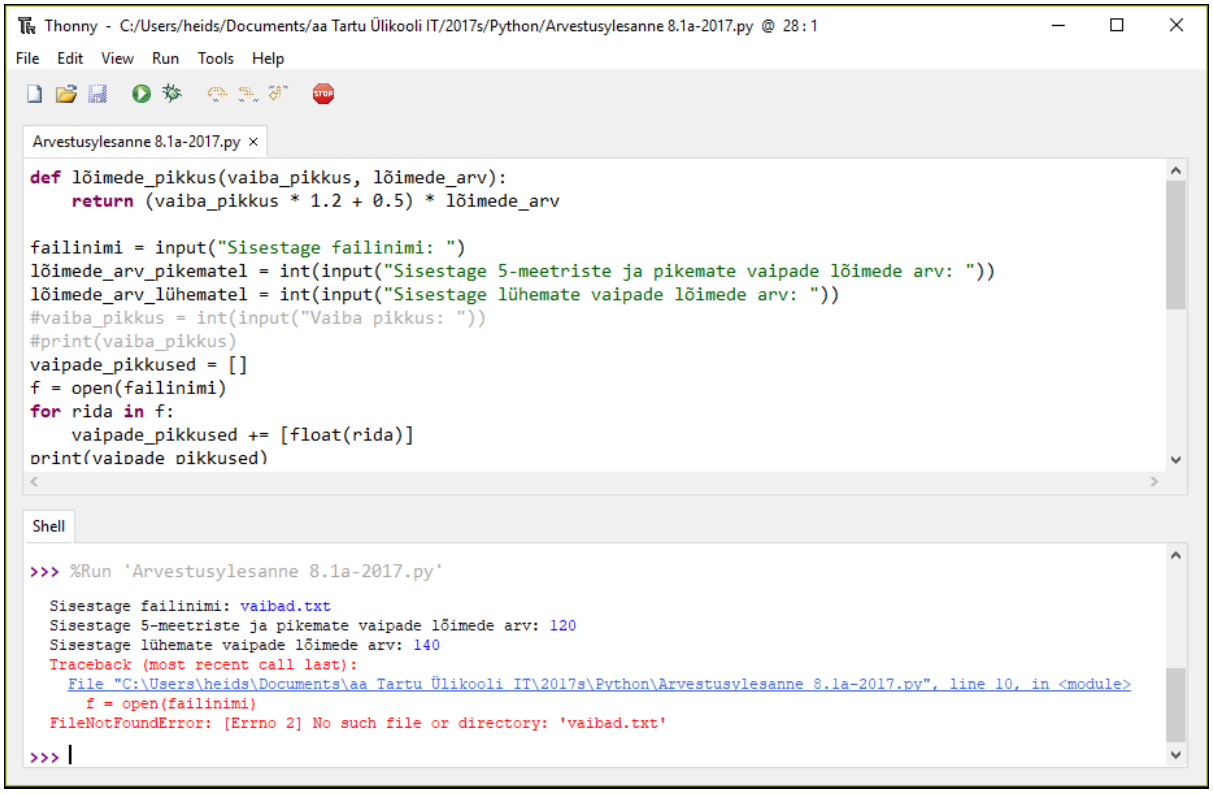
Viis kõige levinumat veateadet olid: *SyntaxError: invalid syntax* (179624), *NameError: name 'NAME' is not defined* (97186), *EOFError: EOF when reading a line* (76026), *SyntaxError: unexpected EOF while parsing* (26097), *IndentationError: unindent does not match any outer indentation level* (20758). Ka siin oli ülekaalukas osakaal süntaksivigadel -

SyntaxError: invalid syntax moodustas 28% kõikidest veateadetest, sageduselt järgmine *NameError: name 'NAME' is not defined* moodustas 15% (Pritchard, 2015).

1.3 Programmeerimiskeskkond Thonny ja logifailid

1.3.1 Thonny tutvustus

Thonny on Pythoni programmeerimiskeskkond, mis on loodud õppimiseks ja õpetamiseks ning mõeldud eelkõige algajatele (joonis 2). Tegu on vabavaralise avatud lähtekoodiga tarkvaraga, mille peamine autor on Tartu Ülikooli õppejõud Aivar Annamaa. Thonnyt on võimalik paigaldada nii Windowsi, Mac OS-i kui ka Linuxi operatsioonisüsteemiga arvutitele ja olemas on ka vastavad juhendid (Annamaa, 2015a; MOOC Programmeerimise alused, 2017a).



Joonis 2. Thonny kasutajaliides

Tarkvara autori eesmärk oli luua keskkond, mis koondaks kõik funktsioonid, mis on vajalikud programmeerimise algõppeks. Samal ajal jälgiti, et kasutajaliides oleks nii lihtne kui võimalik. Thonnyl on olemas kõik tähtsamad funktsioonid, mida pakub Pythoni peamine programmeerimiskeskkond IDLE (*Integrated Development Environment*), kuid on ka omadusi, mida IDLE-il ei ole. Näiteks asub käsurida koodiredaktoriga samas vaates, mis muudab kasutaja tegevuse sujuvamaks ning innustab käsurida väikesteks katsetusteks rohkem

kasutama. Lisaks on käsureal sisend ja väljund ka visuaalselt selgelt eristuvad (Annamaa, 2015b).

Üheks olulisemaks funktsionaalsuseks võib pidada võimalust programmi tööd samm-sammult jälgida ehk siluda (ingl *debug*). Nii on võimalik näha, kuidas programm töötab ja mis järjekorras käsklusi täidetakse, mis on oluliseks abiks ka oma programmi töö kontrollimisel ja vigade leidmisel. Thonny näitab kõiki samme ühes vaates, mis teeb programmi töö jälgimise lihtsamaks.

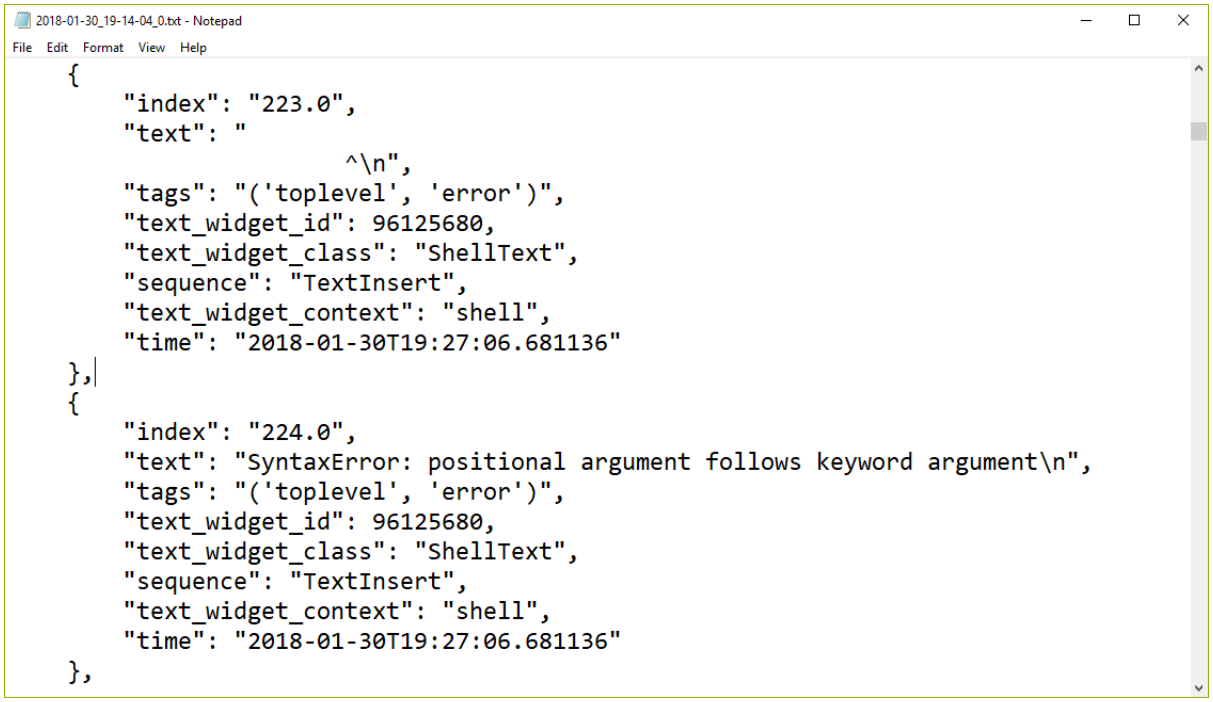
Kui üliõpilaste hulgas viidi pärast Thonny kasutamist läbi küsitlus, siis leidsid paljud, et võimalus programmi samm-sammult läbi vaadata aitas neil programmide konstruktsioonidest paremini aru saada. Samuti kiideti võimalust kasutada käsurida koodiredaktoriga samas vaates (Annamaa, 2015b). Kuna magistritöö fookus on veateadetega seotud tegevustel, on uurimuses vaatluse all ka silumise kasutamine.

Lisaks juba kirjeldatud funktsionaalsusele salvestab Thonny kasutajate tegevused logifaili, mis võimaldab hiljem vaadata programmi loomise ja muutmise protsessi samm-sammult. Kogu programmiga töötamise käigu salvestamine logifaili annab häid võimalusi õppimise uurimiseks. Kuna magistritöö eesmärkidest lähtuvalt on Thonny puhul fookuses just logifailid, siis käsitletakse nimetatud funktsionaalsust veidi täpsemalt järgnevas alapeatükis 1.3.2 ja logifailides sisalduvat infot kirjeldatakse ning analüüsitakse põhjalikumalt alapeatükis 3.1.

1.3.2 Thonny logifailid

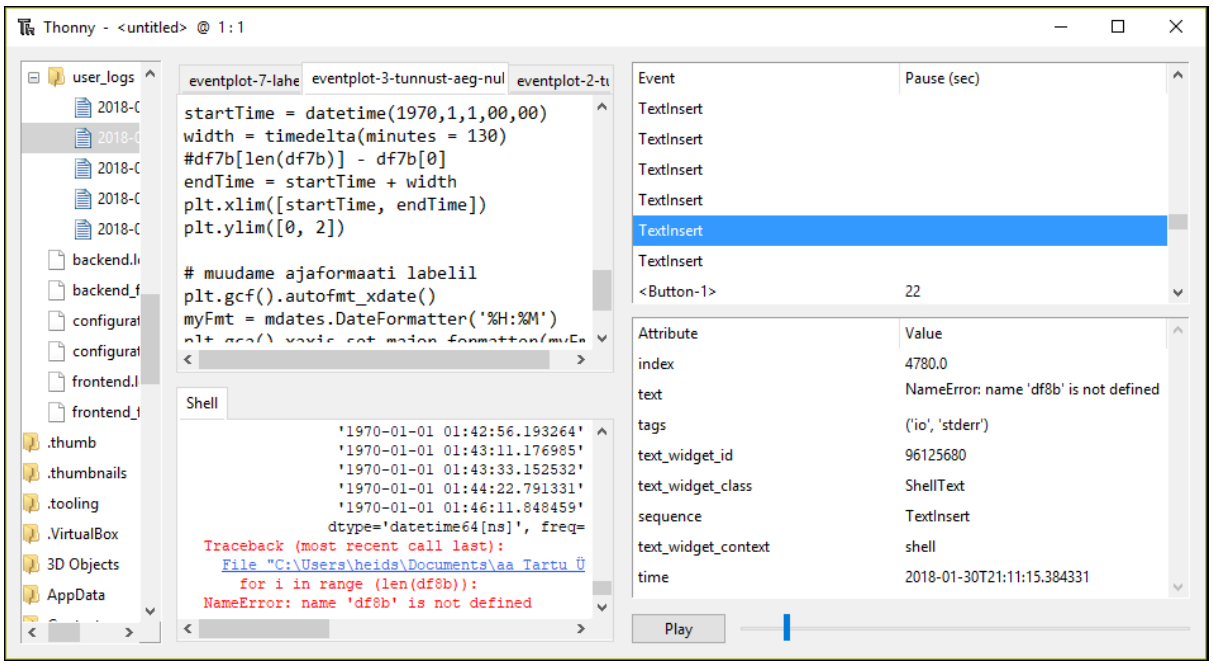
Thonny funktsionaalsus sisaldab info talletamist logifailidesse. Iga kasutussessiooni kohta luuakse üks logifail, mis salvestatakse txt-failivormingus. Failid salvestatakse kohalikule kõvakettale sisseloginud kasutaja profiili alla *user_logs* kausta. Näiteks võib aadress olla C:\Users\kasutaja\.thonny\user_logs, kus *kasutaja* on sisseloginud kasutaja profiili nimi.

Thonny logifailidesse (joonis 3) salvestatakse programmi kasutamise käigus samm-sammult informatsiooni kasutaja tegevuste kohta (nt käivitamised, veateated, silumised, teksti sisestamised, kopeerimised ja kleepimised) ning juurde lisatakse ajatempel (ingl *timestamp*). Thonny logifailid kasutavad lihtsustatud andmevahetusvormingut JSON (ingl *JavaScript Object Notation*) (Annamaa, Hansalu & Tõnisson 2015; Introducing JSON, s.a.; Kodasmaa 2017a).



Joonis 3. Fragment Thonny logifailist

Thonny võimaldab logifaili abil taasesitada kogu programmi loomise protsessi ja näha, milline oli programm erinevatel ajahetkedel. Joonisel 4 on kujutatud hetk kasutussessiooni taasesitamisest, kus on näha ka veateadet.



Joonis 4. Thonny kasutajaliides: kasutussessiooni taasesitamine

Selleks, et logifaili kasutades sessiooni tegevusi taasesitada, tuleb menüüst valida *Tools* > *Open replayer...* ning seejärel avaneb kasutussessiooni taasesitamise kasutajaliides. Kui menüüs ei ole valikut *Open replayer...*, tuleb sisse lülitada *expert mode*, milleks valida

menüüst *Tools* > *Open Thonny data folder...* ja avada fail configuration.ini ning lisada faili algusesse järgmised read:

```
[general]
expert_mode = True
```

Enne faili salvestamist tuleb sulgeda Thonny, et ei salvestataks configuration.ini faili üle juba eelnevalt mälus olnud seadetega.

1.4 Õppimise uurimine programmeerimise algõppes

Kõige rohkem programmeerimisülesannete lahendamise protsessiga seotud uurimusi on algõppe kohta tehtud Java põhjal, Pythoni õppimisega seotud uurimusi on oluliselt vähem. Selles peatükis käsitletakse lisaks uuematele ka mõningaid vanemaid olulisi uurimusi, mida on hilisemates töödes palju refereeritud.

Viimase 15 aasta jooksul on uuringutes üha populaarsemaks muutunud süsteemid, mis salvestavad ja koguvad infot õppijate programmeerimise protsessi kohta. Andmete kogumise metoodika ja detailsus varieerub, ulatudes lõpptulemuse fikseerimisest kuni iga klahvivajutuse salvestamiseni (Helminen, Ihantola & Karavirta, 2013; Vihavainen, Luukkainen & Ihantola, 2014). Kui varasemad süsteemid salvestasid peamiselt infot programmide esitatud versioonide kohta, siis viimastel aastatel on hakatud rohkem tähelepanu pöörama süsteemidele, mis koguvad infot ka lõpptulemusele eelnevate tegevuste kohta (Vihavainen, Luukkainen & Ihantola, 2014). Näiteks uurisid lahendamise protsessi Allevato ja Edwards (2010), kusjuures jälgiti tööde erinevaid esitatud versioone ja leiti, et nõrgemad õppijad testisid programme vähem ja kustutasid parandamise käigus palju koodi (näiteks kogu meetodi).

Õppimise protsessi on uurinud ka Jadud (2006b), kes võttis aluseks kompileerimised ja lõi algoritmi (*Error Quotient (EQ)*), mis võtab arvesse, kui palju on vigu, millised on veateated, kus need asuvad ning kus on tehtud muudatusi. Ta uuris ka, millised olid levinumad veateated Java algõppes ja kui pikk oli kompileerimiste vaheline aeg. Kõige tavalisemate vigade (näiteks puudub semikoolon või sulg) parandamine võttis aega vähem kui pool minutit (51% kompileerimistest toimus vähem kui 30 sekundi jooksul pärast eelmist kompileerimist). Ka Altadmri (2015) uurimus kinnitab, et kõige levinumad süntaksivead on sellised, mida õppijad väga kiiresti parandavad. Jadud (2005) arutleb, et olukorda võiks parandada, kui IDE

näitab õppijale selliste vigade olemasolu kohe, nii et need ei selgu alles kompileerimise käigus. Tänapäevased IDE-d juba sisaldavadki nimetatud funktsionaalsust.

Samuti on veateateid uurinud Becker (2016b), kelle töö rõhuasetus on korduvatel vigadel, tuues välja, et 15 kõige sagedasemat veateadet moodustavad 86,3% kõikidest vigadest. Denny (2012) aga täheldas vigade sagedusi uurides, et süntaksivead on erinevate raskusastmetega. Pythoni veateadete sagedusi käsitlevaid uuringuid vt ptk 1.2.

Lisaks eelnevatele suundadele uurisid nii Denny, Luxton-Reilly ja Carpenter (2014) kui ka Becker (2016a) Java näitel, kas põhjalikumat ja täpsemat kirjeldust sisaldavad veateated aitavad kaasa kompileerimisvigade vähenemisele. Nad jõudsid vastandlikele järeldustele. Denny ei täheldanud olulist erinevust tudengite käitumises, kuid Beckeri uurimus näitas, et korduvate vigade arv vähenes. Pettit, Homer ja Gee (2017) jõudsid Beckeriga sarnasele tulemusele C++ näitel.

Jadud (2005) märgib, et õpetajad on täheldanud, et õppijad kompileerivad sageli programme uuesti, ilma et prooviks vahepeal viga parandada. Ka andmed näitavad seda. Näiteks olid 21% korduvatest semikooloni puudumise vigadest sellised, kus pärast veateadet ei olnud koodi üldse muudetud. Samuti näitasid tulemused, et on õppijaid, kes kompileerivad palju enam kui ülejäänud. Mõned kompileerisid peaaegu 60 korda tund aega kestnud praktikumi jooksul. Autor sellele selgitust ei leidnud.

Samuti täheldas Jadud (2006b), et tugevamad õppijad asuvad probleemse süntaksivea korral kergemini tööle mõne teise programmiosaga ja pöörduvad hiljem probleemse koha juurde tagasi. Jadudi (2006a) järgi on korduvate vigade arv üheks selgemaks indikaatoriks, mis näitab, kui hästi õppija õppetöös edasi jõuab.

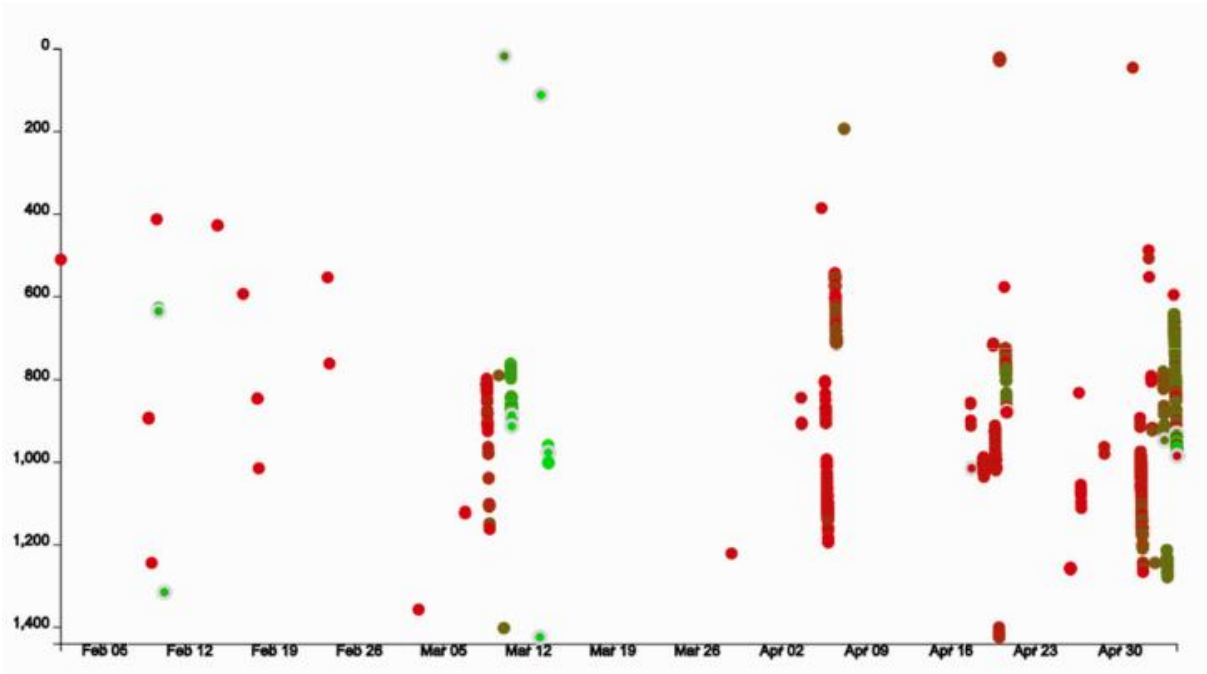
Watson ja Godwin (2013) võtsid samuti veateadete uurimisel aluseks kompileerimised ja mõõtsid lisaks kompileerimiste vahelisele ajale ka aega, mis õppijal kulub konkreetse vea parandamiseks. Nii Jadud (2006b) kui ka Watson ja Godwin (2013) keskendusid algoritmi loomisele, mis võiks eristada tugevamaid ja nõrgemaid õppijaid.

Denny (2011) uuris õppijaid, kes töötasid Java programmidega veebipõhises programmeerimiskeskkonnas. Tulemused näitasid, et suurem osa esitatud programme sisaldasid süntaksivigu. Vihavainen, Helminen ja Ihantola (2014) seevastu jõudsid järeldusele, et kui IDE tõstab süntaksivead esile ja pakub tagasisidet, kuidas neid parandada, siis on üle 90% esitatud programmidest süntaksivigadeta.

Nagu eelnevas näiteski, tuleb aeg-ajalt ette, et uurijad jõuavad vastandlikele tulemustele. Kui paljud uuringud kasutavad ühte või mitut sama õppeasutuse õpperühma, et koguda andmeid, siis Ahadi, Lister, Lal ja Hellas (2018) näitavad oma uurimuses, et see võib olla küllaltki subjektiivne. Nad uurisid vigade sagedusi, parandamiseks kuluvat aega ja vigade muutumist semestri jooksul, kõrvutades andmeid laiapõhjalisemate uuringutega, mis haaravad mitmete erinevate õppeasutuste tulemusi. Nad jõudsid tõdemuseni, et tulemused võivad väga palju varieeruda sõltuvalt sellest, kuidas kursus on läbi viidud.

Spacco, Strecker, Hovemeyer ja Pugh (2005) lähevad kompileerimiste mõõtmise tasandi osas võrreldes eelnevate näidetega detailsemaks: nad arendasid välja süsteemi, mis fikseerib iga salvestamise. Sama detailsusastmega jätkati ka järgmistes töödes. Spacco, Fossati, Stamper ja Rivers (2013) uurisid, millal õppijad eelistasid töötada. Eelistatumaks programmeerimise ajaks kujunes pärastlõuna ja õhtupoolik (kella 4 ja 6 vahel), vähem hilisõhtu. Suur osa tööst tehti 48 tunni jooksul enne tähtaega. Samuti korrelleerus varasem alustamine parema tulemusega. Ajalised eelistused aga on tõenäoliselt väga tihedasti seotud kontekstiga ja võivad seetõttu palju varieeruda.

Balzuweit ja Spacco (2013) tegelesid lisaks programmeerimise protsessi detailsemale uurimisele põhjalikumalt ka tulemuste kuvamise võimalustega. Nad lõid vahendi, mis aitab visualiseerida programmeerimise käigus salvestatud andmeid, paigutades x-teljele aja päevade kaupa ja y-teljele aja minutites arvestatuna südaööst (joonis 5). Lisaks visualiseerisid nad värvide abil koodi korrektsust (roheline korrektne, punane mitte).



Joonis 5. Balzuweit ja Spacco näide koodi korrektsuse visualiseerimisest ajateljel (Balzuweit & Spacco, 2013)

Vihavainen, Luukkainen ja Ihantola (2014) ning Vihavainen, Helminen ja Ihantola (2014) lähevad programmeerimise protsessi uurimises võrreldes varasemate töödega veelgi detailsemaks, jälgides ka seda, kui palju õppijad teksti lisavad, kustutavad või kopeerivad. Nad uurisid, kuivõrd erinev on ülesande lahendamise töömaht sõltuvalt sellest, kas oli olemas eelnev kokkupuude programmeerimisega või mitte. Nimetatud autorid rõhutavad detailsema uurimise vajadust, tuues andmete põhjal esile, et pea pooled õppijad töötavad ülesannetega, mida nad kunagi ei esita, ning mõned töötavad edasi programmidega, mille nad on juba esitanud.

Uurimisrühm lisas programmeerimiskeskkonnale logimise funktsionaalsuse, mis registreerib kõik klahvivajutused ja süsteemis toimunud sündmused. Nii on võimalik detailsemalt uurida, kuidas õppijad koodiga töötavad. Algajate õppijate puhul oli ootuspärane, et esialgu valmistab raskusi Java süntaks. Samas ei leidunud tõendeid, mis kinnitaks, et programmeerimiskeskkonna tundmaõppimine oleks valmistanud raskusi. Autorid arvavad, et algajatele ei ole vaja lihtsustatud programmeerimiskeskkondi. Samuti näitasid tulemused, et õppijad kasutavad kopeerimist ja kleepimist programmeerimise käigus suhteliselt tihti, kuid suurem osa kleebitud koodist pärineb õppijate endi varasematest töödest (Vihavainen, Helminen & Ihantola, 2014).

Kiesmüller, Sossalla, Brinda ja Riedhammer (2010) seevastu uurisid õppijaid, kes kasutasid visuaalset programmeerimiskeskkonda Kara. Nad lõid rakenduse, mis logide põhjal tuvastab erinevaid probleemide lahendamise strateegiaid.

Marceau, Fisler ja Krishnamurthi (2011a, 2011b) kasutasid samuti logimise funktsionaalsust ja vaatasid, milliseid muudatusi õppijad tegid koodis pärast veateadet. Selle põhjal said nad tuvastada, kas veateade aitas õppijal viga parandada. Nad logisid õppijate tegevusi pärast veateadet klahvivajutuse täpsusega kuni järgmise käivitamiseni ning jälgisid ka aega. Iga kord, kui õppija sai veateate, salvestasid nad ka programmi, millega ta töötas. Programmi salvestamised kombinatsioonis klahvivajutustega andsid võimaluse taasesitada õppijate reageerimist veateadetele: näha, millise veateate nad said, kus tegid seejärel parandusi ja kui palju muudatusi tegid, enne kui käivitasid uuesti programmi. Autorite eesmärk oli leida veateated, mis valmistasid raskusi, et parandada veateadete kirjeldusi.

Blikstein (2011) fikseeris samuti õppijate tegevused klahvivajutuse täpsusega, et uurida õppijate käitumist programmeerimisülesannete lahendamisel. Erinevalt eelnevalt mainitud autoritest, keskendus ta eelkõige sellele, et töötada välja meetodeid, kuidas logitud andmeid automaatselt analüüsida ja visualiseerida. Ta kogus andmeid spetsiaalse programmeerimiskeskkonna kaudu, mis salvestas kõik kasutajate tegevused: lisaks muudatustele koodis fikseeriti ka klahvi- ja nupuvajutused jms.

Ta vaatas ka kirjutatud koodi hulka erinevatel ajahetkedel, kompileerimiste vahelist aega ja veateateid. Nende andmete põhjal jaotas Blikstein (2011, 2013) õppijad kolme tüüpi:

1) Kopeerijad-kleepijad (ingl *copy and pasters*). Kood lisandub astmeliselt, vahepeal on perioodid, kus lisandub vähe muudatusi (ajad, mil otsitakse või kohandatakse koodi), aeg-ajalt lisandub järsku palju koodi (kleebitakse leitud või kohandatud kood). Peamiselt omane algajatele programmeerijatele.

2) Iseseisvad (ingl *self-sufficients*). Koodi maht kasvab ühtlaselt, ei kasutata mujalt otsitud koodi kleepimist töösse. Peamiselt omane kogenud programmeerijatele.

3) Segatüüp (ingl *mixed-mode*). Kasutab programmeerides kahe esimese õppijatüübi lähenemist läbisegi.

Autor leiab, et erineva profiiliga õppijad vajavad õppimisel erinevat laadi nõuandeid.

Õppijad jagas tüüpidesse ka Hosseini et al (2014), kes võttis gruppidesse jagamise aluseks programmi konstruktsioonide arvu ning kui palju läbis programm teste, arvestades ka kompileerimiste ja salvestamiste arvu. Autori arvates ei ole õppimistüüp püsiv, vaid võib kogemuse ja oskuste kasvades muutuda. Grupid olid järgmised:

1) Ehitajad (*builders*), kes lisavad järk-järgult uusi konstruktsioone ja nii läbib programm ka järjest rohkem teste.

2) Massöörid (*massagers*). Üldiselt töötavad sarnaselt ehitajatega, kuid neil on ka perioode, kus teevad ainult pisimuudatusi.

3) Skulptorid (*reducers*). Kirjutavad kogu programmi valmis ja hakkavad seejärel selle struktuuri parandama või leiavad vajalikule sarnase programmikoodi ja hakkavad seda kohendama, lisades vajalikku ja eemaldades ebavajalikku.

4) Vaevlejad (*strugglers*). Neil läheb kaua aega ja vaeva, et programm hakkaks teste läbima.

Hosseinei et al (2014) liigituse võttis analüüsi aluseks magistriõppe lõputöös Hansalu (2015), kes tõlkis õnnestunult eesti keelde ka gruppide nimed.

Kui Blikstein (2011) analüüsis veateateid ja kompileerimisi koos ajalise määranguga, siis joonistus välja kolm faasi. Esimeses ehk uurimise faasis on vähe ebaõnnestunud kompileerimisi, millele järgneb aktiivne koodiloome faas, kus on palju ebaõnnestunud kompileerimisi. Kolmandas faasis on peamiselt väikesed parandused, mille käigus on taas vähem veateateid.

Kuigi õppeprotsessi detailne uurimine on viimasel ajal hoogustunud, on õppijate käitumismustreid käsitlevaid uurimusi veel küllaltki vähe. Peamiselt on fookus olnud suunal, mis püüab kaardistada veateateid, mis vajavad täpsemat kirjeldamist.

2. Uurimuse materjal ja metoodika

Selles peatükis kirjeldatakse, kuidas ja milliste vahenditega analüüsiti õppijate käitumismustreid programmeerimisülesande lahendamisel. Samuti vaadeldakse, kuidas valiti välja logifailid, kust infot koguti. Logifailidega seotud infot ja tööd andmetega kirjeldatakse suure detailsusega, et järgmistel uurijatel oleks olemas informatsioon, mida kasutada.

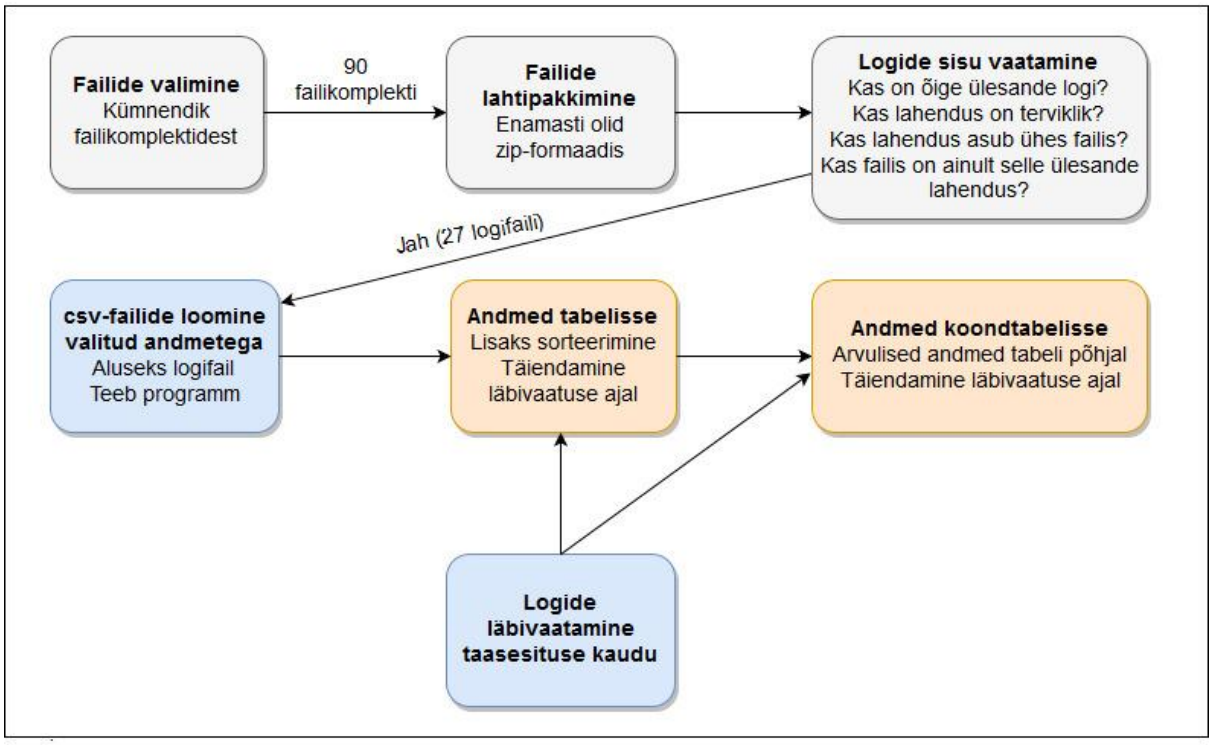
Magistritöös seati eesmärgiks logifailide põhjal leida vastused järgmistele küsimustele:

1) Millist informatsiooni on võimalik Thonny logifailidest koguda õppijate tegevuse kohta?
2) Millised on õppijate erinevad käitumismustrid programmeerimisülesannete lahendamisel MOOC-i "Programmeerimise alused" kokkuvõtva arvestusülesande näitel?
3) Milliseid õppijatüüpe saab eristada ülesande lahendamise käitumismustrite alusel?

Küsimustele otsiti vastuseid logifailides sisalduva informatsiooni põhjal. Nimelt talletab Tartu Ülikooli õppejõu Aivar Annamaa väljatöötatud Pythoni programmeerimiskeskkond Thonny programmeerimise käigus logifaili samm-sammult detailset informatsiooni kasutaja tegevuse kohta (täpsemalt on Thonny funktsionaalsusest ja logifailidest juttu alapeatükis 1.3 ning logifailide info põhjalikum analüüs alapeatükis 3.1).

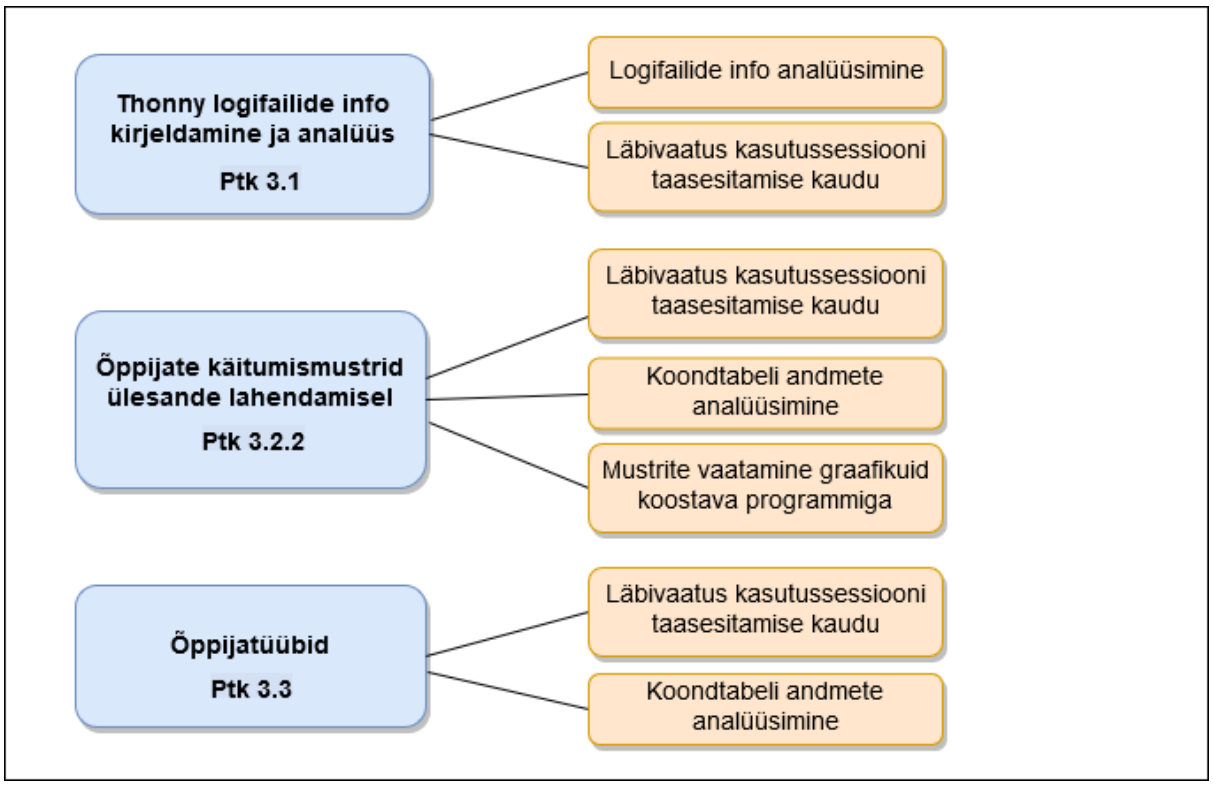
Magistritöös kasutati õppeprotsessi uurimiseks Thonny logifaile Tartu Ülikooli MOOC-ilt "Programmeerimise alused". Nimelt kasutati 2017. aasta sügis-talvel toimunud kursuse arvestusülesande 8.1a logisid, mis olid juba olemas (nimetatud kursusel koguti õppijatelt arvestusülesande logifaile). Ülesanne hõlmas kursuse peamisi teemasid, täpne tekst on peatükis 3.2.1.

Kasutati ainult neid logisid, mis asusid ühes failis ning samas failis ei olnud teiste ülesannete lahenduskäike. Järgnev skeem (joonis 6) kirjeldab ülevaatlikult, kuidas toimus failide selekteerimine. Lisaks on kirjeldatud sellele järgnenud tegevusi: valitud andmete kirjutamine programmi abil logidest csv-failidesse (iga lahendaja kohta üks csv-fail), csv-failide info koondamine andmete tabelisse (lisa 3, kus on kõikide lahenduste info koos), summeeritud andmete kogumine koondtabelisse (lisa 4) ja läbivaatuse info lisamine. Protsesse on täpsemalt kirjeldatud allpool.



Joonis 6. Töö failidega ja andmete kogumine

Paralleelselt andmete kogumise infoga kirjeldatakse ka analüütilisi osi (kokkuvõtvalt joonisel 7). Täpsemalt on meetodeid kirjeldatud allpool.



Joonis 7. Andmete analüüsi skeem

Analüüsitavad lahendused valiti mitmes etapis. Esialgu võeti logi esitanud osalejate nimede tähestikulises järjekorras iga kahekümnes (kokku 45). Kuna aga nende hulgas ei olnud piisavalt selliseid, kus logifailis oli täpselt ühe ülesande terviklahendus, siis võeti samasugusel alusel veel 45. Nii saadi 90 lahenduskäigu andmed, mis peamiselt olid esitatud zip-failidena, mis sisaldasid ühte või mitut tekstifaili. Oli ka rar- ja 7z-formaadis kokkupakitud faile ja logisid, mis olid kokkupakkimata. Mõnel juhul sisaldas zip-fail omakorda teisi zip-faile. Kokkupakitud failid pakiti lahti ning seejärel vaadati läbi, neist 76 sisaldas õige ülesande logi ning 76-st 44 olid ühes failis, millest 13 ei sisaldanud terviklahendust, neli lahendas ühes logis rohkem kui ühte ülesannet. Ühel lahendajal olid terve kursuse ülesannete lahendused ühes logis, mis tähendab, et ta ei sulgenud terve kursuse jooksul kordagi Thonnyt. Valimisse võeti tehnilise töö vähendamise eesmärgil need logifailid, kus ülesande terviklahendus asus ühes failis ning samas failis ei olnud teiste ülesannete lahenduskäike. Selliseid faile oli 27.

Selleks et töötada välja metoodika logifailis sisalduva info kasutamiseks programmeerimise õppijate käitumismustrite analüüsimiseks, tuli esmalt põhjalikult analüüsida logifailide ülesehitust ja seal sisalduvat informatsiooni (logifailide info kirjeldamine ja analüüs on peatükis 3.1). See analüüs oli hiljem aluseks ka järgmiste sammude planeerimisel ja logidest vajalikku informatsiooni koguva programmi koostamisel. Nimelt tuli välja selgitada, mida ja milliste reeglite alusel saab logifailidest kätte masinlikult ning millist olulist informatsiooni oli vaja hankida käsitsi läbivaatamise kaudu, kasutades Thonny kasutussessiooni taasesitamise funktsionaalsust.

Magistritööga seoses kirjutati kaks programmi:

1) Valitud andmeid logist tabelisse kirjutav programm (loodi juhendaja abiga) (lisa 2).
2) Graafikuid koostav programm (loodi iseseisvalt) (lisa 1), mida kasutati lisaks oluliste andmete magistritöös visualiseerimisele ka erinevate andmekombinatsioonidega katsetamiseks, et näha võimalikke iseloomulikke käitumismustreid. Programm loodi Pythonis, kasutades andmetöötluseks ja -analüüsiks mõeldud moodulit Pandas ja graafikute kuvamise moodulit Matplotlib.

Logist valitud andmeid tabelisse kirjutav programm loeb logifailist info sisse ja genereerib iga logi kohta csv-faili, mis sisaldab logist saadud selle töö seisukohast olulisi andmeid. Programmi abil saadi logifailidest veateated koos ajatempliga, aeg veateatest järgmise käivitamiseni, käivitamised koos ajatempliga, programmikoodi tähemärkide arv iga

käivitamise hetkel, kas käivitamisele järgnes või ei järgnenud veateade; silumised ja kas silumisele järgnes või ei järgnenud veateade, ülesande alustamise ja lõpetamise aeg ning kestus. Kuna ühes logifailis võib olla informatsiooni mitme ülesandega seotud tegevuste kohta (näiteks käivitas õppija Thonnys varasemate ülesannete lahenduste programme), võeti tähemärkide arvu, ülesande algus- ja lõpuaja fikseerimisel arvesse ka võtme "text_widget_id" väärtust, mis on iga failiga programmiaknas töötamisel erinev (täpsemalt saab logifailide võti-väärtus paaride süsteemi kohta lugeda peatükist 3.1).

Lisaks vaadati logid Thonny kasutussessiooni taasesitamise funktsionaalsust kasutades läbi. Nimelt on õppijate käitumises mustreid, mida oli võimalik eelkõige märgata läbivaatuse käigus. Samuti korrigeeriti läbivaatuse käigus programmi abil saadud andmeid, kui saadi näiteks teada, et õppija käivitas töö käigus Thonnys ka teisi programme. Ka ei ole masinlikult lihtne tuvastada, kas õppija proovis viga parandada või tegi pärast veateadet midagi muud. Lisaks on veateate puhul oluline, kas õppija parandas vea. Kui järgmisel käivitamisel veateadet ei tulnud, ei pruukinud viga olla parandatud, õppija võis vigase koha näiteks välja kommenteerida. Masinlikult kogutud andmete hulgas oli ka aeg veateatest järgmise käivitamiseni, käivitamise edukus ja tähemärkide arv käivitamise hetkel. Need otsustati analüüsist välja jätta, kuna läbivaatusel selgus, et need andmed vajavad suuremat korrigeerimist ja magistritöö raames loodud programm täiustamist.

Enne läbivaatust võeti programmi genereeritud csv-failidest iga lahendaja kohta informatsioon ja pandi käsitsi kokku ühte tabelisse (fragment joonis 8, terviktabel lisa 3) ning sorteeriti lahendajate kaupa ajatempli järgi. Lisaks loodi tabelisse tulbad, kuhu märkida info, mille annab läbivaatus. Nii vaadati läbivaatuse käigus veateatele järgnevaid tegevusi, valides järgmiste valikute hulgast: parandas vea, veateadet ei tulnud; parandas vea, tuli teistsugune veateade; proovis viga parandada, sama veateade tuli uuesti; proovis viga parandada, tuli teistsugune veateade; ei teinud mitte midagi, sama veateade tuli uuesti; hakkas tegema midagi muud, sama veateade tuli uuesti; hakkas siluma; kommenteeris osa koodi välja, veateadet ei tulnud; kommenteeris osa koodi välja, tuli teistsugune veateade; kustutas või lõikas olulise osa koodi ära; muutis koodi, nii et veateadet ei tulnud, aga viga ei ole leitud; käsuaknasse sisestamise viga; käivitas muu faili.

ID		Käivitamised ja veateated	Täpne fraas	Aeg	Aeg veateatest järgmise käivitamiseni	Mitu tähemärki koodi, kui käivitab
16		Käivitamine		2017-12-10T16:56:03.553860		213
16		Käivitamine		2017-12-10T16:56:51.571439		262
16	2	NameError: name 'vaiba_pikkus' is not defined		2017-12-10T16:56:51.684875	00:03:17	
16		Käivitamine		2017-12-10T17:00:08.982335		280
16	3	NameError: name 'vaiba_pikkus' is not defined		2017-12-10T17:00:09.151841	00:00:14	
16		Käivitamine		2017-12-10T17:00:22.657388		280
16	4	NameError: name 'vaiba_pikkus' is not defined		2017-12-10T17:00:22.809795	00:00:35	
16		Käivitamine		2017-12-10T17:00:57.738058		255
16	5	TypeError: lõimede_pikkus() missing 2 required positional arguments: 'vaiba_pikkus' and 'lõimede_arv'		2017-12-10T17:00:57.901595	00:00:48	

Joonis 8. Fragment andmete tabelist (terviktabel lisa 3)

Andmete valikul olid aluseks varasemad uurimused, mis on just neid indikaatoritena oluliseks pidanud. Näiteks on kirjutatud koodi hulka erinevatel ajahetkedel, kompileerimiste vahelist aega ja veateateid koos uurinud Blikstein (2011, 2013). Marceau, Fisler ja Krishnamurthi (2011a, 2011b) aga jälgisid, milliseid muudatusi tegid õppijad pärast veateadet ja kui palju kulus aega. Samuti on autoreid (Jadud, 2005), kes on täheldanud, et osa õppijad kompileerivad palju enam kui ülejäänud. Kindlasti on oluline näitaja ka korduvate vigade arv, sest seda on peetud üheks selgemaks indikaatoriks, mis näitab, kui hästi õppija õppetöös edasi jõuab (Jadud, 2006a). Läbivaatuse valikutesse sai lisatud ka võimalus, et õppija ei teinud pärast veateadet mitte midagi. Ka kirjanduses on sellest variandist juttu (Jadud, 2005). Põhjalikumalt on varasemaid uurimusi käsitletud alapeatükis 1.4.

Läbivaatust peeti oluliseks ka õppijate erinevate üldisemate käitumismustrite märkamiseks (nt kui palju avab faile; kas lahendab ülesannet varasema ülesande koodi kohendades; kas käivitab teisi ülesandeid; kas käivitab esimest korda siis, kui funktsioon on kirjutatud jne), mida kõrvutati hiljem masinlikult saadud andmetega. Samuti jälgiti, kas leidub infot, millest võiks abi olla õppijatele ja õpetajatele. Tähelepanekud kanti tabelitesse (lisa 3 ja lisa 4) ja iga järgneva lahendaja logi läbi vaadates jälgiti nende esinemist ja lisati uusi. Kuna osa käitumismustrite kordumine selgus alles viimaste logide läbivaatamisel, vaadati kõik logid üldiste käitumismustrite seisukohast läbi kaks korda. Kui kogu informatsioon oli koos, töödeldi ja võrreldi andmeid, et leida seoseid.

Juba läbivaatuse käigus oli näha, et õppijate tegevus erines selle poolest, kas nad töötasid järk-järgult või mitte, s.t kas nad kirjutasid programmi osade kaupa ja vahepeal ka testisid selle tööd või testisid esimest korda alles siis, kui olid suurema osa lahendusest valmis kirjutanud. Osade kaupa lahendajad testisid programmi esimest korda pärast funktsiooni kirjutamist. Kuna kõikide lahenduste kohta koguti mitmekülgset informatsiooni, siis järgnevalt vaadati funktsiooni testimist või mittetestimist ülejäänud tunnuste kontekstis.

Selleks, et analüüsida õppijate käitumismustreid lahendamise sujuvuse perspektiivis, valiti välja sisuliselt olulised tunnused, mis on peamiselt omased õppijatele, kellel kulub ülesande lahendamiseks rohkem aega. Põhialuseks võeti korduvad veateated, mille sagedust on peetud üheks selgemaks indikaatoriks, mis näitab, kui hästi õppija õppetöös edasi jõuab. Sellisel seisukohal on näiteks Jadud (2006a). Selles magistritöös kirjeldatud käitumisviise ajakuluga kõrvutades leiti, et lisaks veateadete ja korduvate veateadete suuremale hulgale, tuli aeglasematel lahendajatel hiljem rohkem uuesti ka varem esinenud veateateid. Samuti hinnati oluliseks näitajaks teiste ülesannete lahenduste kasutamist ülesande lahendamise ajal.

Seega vaadati, kui palju oli korduvaid veateateid ning kui palju oli olukorda, kus veateatele järgnes teistsugune veateade ja seejärel tuli varasem uuesti. Samuti vaadati, kui palju avasid õppijad ülesande lahendamise ajal teiste ülesannete lahenduste faile.

3. Õppijate käitumismustrite analüüs logifailide põhjal

3.1 Thonny logifailide info kirjeldamine ja analüüs

Selles alapeatükis tuleb juttu Thonny logifailidest, nendes sisalduvatest kirjetest ja täpsemast informatsioonist, samuti aspektidest, millega tuleb arvestada logifailidest info kogumisel. Vaatluse all on ka võimalused leida kirjetest andmeid, mida on võimalik kasutada õppeprotsessi uurimiseks. Thonny logifailide funktsionaalsusest üldisemalt on kirjutatud peatükis 1.3.2.

Varem on Thonny logifailides sisalduvat infot kirjeldanud ja analüüsinud Pedel (2016), võttes ülesandeks kirjeldada ülevaatlikult väga erinevat tüüpi kirjeid, jagades need viieks tüübiks: Thonny programmiaknaga seotud kirjed, programmikoodiga seotud kirjed, failidega seotud kirjed, programmi käivitamisega seotud kirjed ja käskudega seotud kirjed. Selles töös kitsendatakse kirjete valikut, s.t keskendutakse eelkõige logifailides sisalduvatele kirjetele, mis on seotud programmikoodi loomise ja teksti kuvamisega, ning võimalustele eristada, kus teksti muutmine või kuvamine toimus. Pedeli (2016) liigituse alusel kuuluvad valitud kirjed peamiselt programmikoodiga ja programmi käivitamisega seotud kirjete hulka.

Thonny iga kasutussessiooni kohta luuakse logifail, mis salvestatakse txt-failivormingus, kasutades andmevahetusvormingut JSON (ingl *JavaScript Object Notation*). Kasutussessioonina käsitletakse Thonny kasutamist alates avamisest kuni sulgemiseni. Näiteks kui õppija avab ja sulgeb Thonny programmeerimiskeskkonda ühe ülesande lahendamise jooksul korduvalt, siis on tegu mitme kasutussessiooniga ja ühe ülesande kohta tekib mitu logifaili. Kui aga õppija lahendab mitu ülesannet, ilma et vahepeal Thonny't sulgeks, tekib mitme ülesande kohta üks logifail.

Logifailid sisaldavad kirjeid paljude eri tüüpi sündmuste kohta, näiteks teksti sisestamised, kustutamised, kopeerimised, kleepimised, faili avamised, programmi salvestamised, käivitamised, silumised ja veateadete kuvamised. Seda infot on võimalik masinlikult koguda. Tegevuste detailsemaks vaatlemiseks on vajalik vahet teha, mis on kasutaja aktiivne tegevus ja millal käivitab sündmuste ahela programm ise. Näiteks saab kasutaja programmi hetkeseisu salvestada, kuid käivitamisel salvestab Thonny selle ise, kui pärast eelmist salvestamist on tehtud muudatusi. Seega tuleb arvestada, et logi põhjal ei ole alati võimalik vahet teha, kas kasutaja salvestas programmi enne käivitamist või mitte. Samuti vajab

tähelepanu, kus tegevus toimus: on tegevusi, mis võivad toimuda nii Thonny programmi- kui ka käsuaknas (näiteks teksti sisestamine).

Vastavalt JSON andmevahetusvormingule koosnevad logifailis sisalduvad kirjed võti-väärtus paaridest, mis on eraldatud komadega. Iga kirje on looksulgudes ning kirjed on omavahel samuti eraldatud komadega. Erinevatel kirjetüüpidel on erinev arv võti-väärtus paare. Võti-väärtus paarid varieeruvad, kuid kõikides kirjetes on olemas võti "sequence", millele vastav väärtus näitab tegevust (nt "TextInsert", "TextDelete", "<Button-1>", "Open", "<FocusIn>", "<FocusOut>"), ja võti-väärtus paar tegevuse ajatempli kohta, mis võib olla näiteks "time": "2017-12-06T17:42:21.856677". Kuna aja info on olemas kõikide sündmuste kohta, on võimalik arvutada ka ajavahemikke erinevate sündmuste vahel. Masinlikult on võimalik leida ka tähemärkide arvu sündmuste (näiteks käivitamiste) hetkel.

Tegevus võib toimuda kas programmi- või käsuaknas. Näiteks ei tähenda "sequence": "TextInsert" ainult teksti trükkimist programmiaknasse (joonis 9), vaid ka teksti lisamist või lisandumist käsuaknasse.


```
{
    "index": "22.37",
    "text": "p",
    "tags": "None",
    "text_widget_id": 77816144,
    "text_widget_class": "CodeViewText",
    "sequence": "TextInsert",
    "time": "2017-12-06T17:41:49.342833"
},
```


Joonis 9. Teksti programmiaknasse sisestamise kirje

Seega vajavad tähelepanu nii teksti asukoht (programmi- või käsuaken) kui ka lisamise viis (kas kasutaja lisab selle või hoopis programm kuvab ekraanile). Näiteks on ka veateate kuvamise kirje puhul võtme “sequence” väärtuseks "TextInsert" (joonis 10), mis näitab, et veateate tekst lisandus. Tegevuse toimumist programmi- või käsuaknas näitab võti "text_widget_class", mille väärtus on "CodeViewText", kui tegevus toimub programmiaknas (joonis 9) või "ShellText", kui tegevus toimub käsuaknas (joonis 10).

```
{
    "index": "39.0",
    "text": "TypeError: unsupported operand type(s) for +=: 'int' and 'tuple'\n",
    "tags": "('io', 'stderr')",
    "text_widget_id": 70234352,
    "text_widget_class": "ShellText",
    "sequence": "TextInsert",
    "text_widget_context": "shell",
    "time": "2017-12-06T17:48:48.966911"
},
```

Joonis 10. Tüübivea kirje

Lisaks on tekstiga seotud kirjes võti “text_widget_id”. Juhul, kui kasutaja töötab terve kasutussessiooni vältel ainult ühe Pythoni failiga ja teistes failides muudatusi ei tee, siis on ühes logifailis kasutusel üks “text_widget_id” programmi- ja teine käsuakna töö jaoks. Kui töö toimub veel mõne Pythoni failiga, siis sealset tegevust on võimalik teiste failidega seotud tegevustest eristada “text_widget_id” väärtuse abil, mis on erinev.

Kui tegevus toimub käsuaknas (“text_widget_class”: “ShellText”), saab kirjete järgi eristada ka seda, kas programm väljastas midagi (näiteks väljastati kasutajale küsimus või programmi töö tulemus) või sisestas kasutaja ise teksti. Esimesel juhul on võtme “tags” üheks väärtuseks “stdout” (joonis 11), teisel juhul “stdin” (joonis 12). Veateadete kuvamisel on “tags” üheks väärtuseks “error” (kui veatüüp on süntaksiviga) või “stderr” (muud veatüübid).


```
{
    "index": "110.0",
    "text": "Sisestage 5-meetriste ja pikemate vaipade l\u00f5imede arv: ",
    "tags": "('io', 'stdout')",
    "text_widget_id": 70234352,
    "text_widget_class": "ShellText",
    "sequence": "TextInsert",
    "text_widget_context": "shell",
    "time": "2017-12-06T18:11:34.538233"
},
```

Joonis 11. Kasutajalt küsimise kirje


```
{
    "index": "110.55",
    "text": "1",
    "tags": "('io', 'stdin')",
    "text_widget_id": 70234352,
    "text_widget_class": "ShellText",
    "sequence": "TextInsert",
    "text_widget_context": "shell",
    "time": "2017-12-06T18:11:35.954657"
},
```

Joonis 12. Teksti käsuaknasse sisestamise kirje

Võtme “index” väärtus näitab, mitmendasse ritta ja mitmendale positsioonile reas tekst sisestati. Näiteks tähendab joonisel 7 “index”: “22.37”, et tekst “p” lisati 22. ritta positsioonile 37, kusjuures positsioonide arvestamine algab kohalt 0. Joonisel 8 on näha veatüüp ja täpne veateate tekst, mis lisandus käsuaknasse (text_widget_class”: “ShellText”) 39. ritta alates positisioonist 0 (“index”: “39.0”).

Kustutamise kirjes (joonis 13) aga on võtmed “index1” ja “index2”, millest esimese väärtus näitab, millisel positsioonil oli esimene tähemärk, mis kustutati. “Index2” väärtus näitab, millisel positsioonil oli viimane tähemärk, mis kustutati. Kui aga “index2” väärtuseks on “None”, siis kustutati korraga üks tähemärk. Kui teksti sisestamise kirjetes sisaldub ka tekst ise (võtme “text” väärtus), siis kustutamiste puhul ei ole logifailis infot, millised tähemärgid täpselt kustutati.


```
{
    "index1": "2.51",
    "index2": "None",
    "text_widget_id": 77816144,
    "text_widget_class": "CodeViewText",
    "sequence": "TextDelete",
    "time": "2017-12-06T17:15:30.885465"
},
```


Joonis 13. Kustutamise kirje logifailis

Ühe ja sama tegevusega võib logifailis olla seotud mitmeid kirjeid. Näiteks kui trükkida klaviatuurilt programmiaknasse teksti, luuakse teksti sisestamise kirje iga tähe kohta eraldi. Seega võib logifailis iga tähe trükkimine olla kajastatud eraldi tegevusena. Samuti on näiteks süntaksivea (ingl *SyntaxError*) puhul järjest seitse kirjet, mille võti “tags” omab väärtust “error” (ühe veateate kuvamise käigu kohta on logifailis seitse kirjet), kusjuures neist kuues sisaldab veatüüpi (joonis 14). Tekstiline viide vea täpsemale asukohale asub neljandas kirjes (joonis 15) ja rea number, kus viga asub, on osa võtme “text” väärtusest teises kirjes.

```
{
    "index": "11.0",
    "text": "SyntaxError: invalid syntax\n",
    "tags": "('toplevel', 'error')",
    "text_widget_id": 70234352,
    "text_widget_class": "ShellText",
    "sequence": "TextInsert",
    "text_widget_context": "shell",
    "time": "2017-12-06T17:42:21.856677"
},
```


Joonis 14. Süntaksivea kirje


```
    {
        "index": "9.0",
        "text": "    l\u00f5imeniidi_pikkus += l\u00f5imede_pikkus
(rida2, l\u00f5imede arv)\n",
        "tags": "('toplevel', 'error')",
        "text_widget_id": 70234352,
        "text_widget_class": "ShellText",
        "sequence": "TextInsert",
        "text_widget_context": "shell",
        "time": "2017-12-06T17:42:21.856677"
    },
```


Joonis 15. Süntaksivea täpsustava teksti kirje

Samas on veatüüpe, mille võti "tags" omab väärtust "stderr" (näiteks tüübiviga (ingl *TypeError*), nimeviga (ingl *NameError*) ja väärtuse viga (ingl *ValueError*)) ning veateate kuvamise käigu kohta on logifailis kuus kirjet, kusjuures veatüüpi sisaldab samuti kuues kirje (joonis 16).


```
{
    "index": "75.0",
    "text": "NameError: name 'l\u00f5imede_arv' is not defined\n",
    "tags": "('io', 'stderr')",
    "text_widget_id": 70234352,
    "text_widget_class": "ShellText",
    "sequence": "TextInsert",
    "text_widget_context": "shell",
    "time": "2017-12-06T18:06:15.267293"
},
```


Joonis 16. Nimevea kirje

Kleepimise puhul luuakse logifaili ühe tegevuse kohta kaks omavahel seotud ja täpselt sama ajatempliga kirjet (joonis 17), millest esimene annab informatsiooni, milline tekst ja kuhu sisestati. Teine kirje aga näitab, et teksti sisestamise viisiks oli kleepimine. Võtme "index" väärtus "6.0" joonisel 17 näitab, et sisestati 6. ritta alates positsioonist 0. Tekst sisestati

programmiaknasse (“text_widget_class”: CodeViewText”) ja sisestamise viisiks oli kleepimine (teises kirjes “sequence”: “<<Paste>>”)


```
{
    "index": "6.0",
    "text": "f = open(failinimi)\nfor rida in f:",
    "tags": "None",
    "text_widget_id": 77816144,
    "text_widget_class": "CodeViewText",
    "sequence": "TextInsert",
    "time": "2017-12-06T17:20:33.655258"
},
{
    "widget_id": 77816144,
    "widget_class": "CodeViewText",
    "text_widget_id": 77816144,
    "text_widget_class": "CodeViewText",
    "sequence": "<<Paste>>",
    "time": "2017-12-06T17:20:33.655258"
},
```


Joonis 17. Kleepimise kirjed

Kui luua programme, mis kasutavad logifailide informatsiooni, siis peab arvestama, et võti-väärtus paaride paigutus kirjes võib olla erinev sõltuvalt Thonny versioonist, mida õppija on kasutanud. Teksti sisestamise kirje võti-väärtus paaride järjekorra erinevus on näha joonisel 18. Lisaks on joonisel näidatud varasemas versioonis võtme “text_widget_class” väärtus “Text” ja hilisemas “CodeViewText”.


```
{
    "text_widget_id": 47485072,
    "text_widget_class": "Text",
    "index": "12.0",
    "sequence": "TextInsert",
    "time": "2016-03-12T19:25:20.996768",
    "tags": "None",
    "text": "p"
}
```

```
{
    "index": "1.0",
    "text": "f",
    "tags": "None",
    "text_widget_id": 77816144,
    "text_widget_class": "CodeViewText",
    "sequence": "TextInsert",
    "time": "2017-12-06T17:14:20.317742"
},
```


Joonis 18. Teksti sisestamise kirjed: vasakul aastast 2016 (Pedel, 2016) ja paremal 2017 Thonny versiooniga 2.1.16

Logid sisaldavad väga detailset informatsiooni ja ühe sündmuse kohta võib olla mitmeid kirjeid. Näiteks esineb sõne *%Run* ühe käivitamise korral tavaliselt kolmes kirjes. Üks võimalus sündmuste loendamiseks on kasutada võtme “text” väärtusi. Konkreetse käivitamise

puhul on täpselt ühes kirjes võtme “text” väärtuse alguses *%Run* (joonis 19). Silumise korral algab vastav väärtus *%Debug*.


```
{
    "index": "2.4",
    "text": "%Run 'Arvestusylesanne 8.1a-2017.py'\n",
    "tags": "('automagic', 'toplevel', 'command')",
    "text_widget_id": 70234352,
    "text_widget_class": "ShellText",
    "sequence": "TextInsert",
    "text_widget_context": "shell",
    "time": "2017-12-06T17:22:07.372020"
},
```


Joonis 19. Käivitamise kirje

Veateadete puhul algab võtme “text” väärtus sõnega *Error:* (sõne *Error* koos kooloniga on unikaalne, s.t veateateid on võimalik leida ka sõneotsinguga). Unikaalse sõne kaudu saab kirjetest leida veateateid ka nende tüübi järgi, sõne on vastavalt *NameError:*, *TypeError:*, *ValueError:* jne (joonis 20).


```
{
    "text": "ValueError: could not convert string to float: 'None'\n",
    "text_widget_context": "shell",
    "sequence": "TextInsert",
    "text_widget_class": "ShellText",
    "tags": "('io', 'stderr')",
    "text_widget_id": 41234896,
    "time": "2017-12-06T21:54:29.282097",
    "index": "236.0"
},
```


Joonis 20. Väärtuse vea kirje

Veateadete analüüsimisel on vajalik tähele panna, et veateateid ei kuvata käsuaknas ainult pärast käivitamist, vaid näiteks ka silumise käigus.

3.2 Õppijate käitumismustrid ülesande lahendamisel

3.2.1 Ülesanne

Analüüsi aluseks oli 2017. sügis-talvel toimunud Tartu Ülikooli MOOC-i “Programmeerimise alused” arvestusülesanne. Kursusel oli igaks nädalaks vaja lahendada ülesandeid. Viimasel nädalal tuli lahendada arvestusülesanne või teha loovtöö. Ka arvestusülesannet võisid kursusel osalejad teha vabalt valitud ajal, samuti ei olnud piiratud lahendamise kestus. Lisaks arvestusülesande lahendusele või loovtööle tuli veel esitada

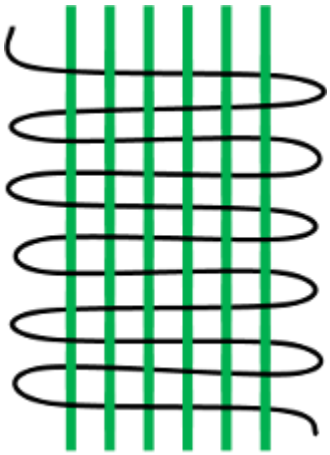
vastava ülesande lahendamise protsessi kirjeldus (eelkõige Thonny logifailina). Ülesannet oli kodulehel kirjeldatud järgmiselt (MOOC Programmeerimise alused, 2017b):

8.2 Arvestusülesanne

Kohustuslikult tuleb lahendada (8.1a ja 8.2a) või (8.1b ja 8.2b).

KONTROLLÜLESANNE 8.1a ARVESTUSÜLESANNE

Tõenäoliselt on paljud näinud kaltsuvaipu, kuid võib-olla pole mõelnud sellele, kuidas neid valmistatakse. Kaltsuvaipu kootakse spetsiaalsetel telgedel, millele kinnitatakse hulk tugevast materjalist niite ehk lõimi, mille vahele hakatakse põimima kaltsuribasid. Lõimed on järgneval joonisel kujutatud rohelisena. Meie eesmärk on vajaliku lõimeniidi kogupikkuse leidmine. Kuna lõimed on telgedel pinges, siis pärast valmimist tõmbub vaip mõnevõrra kokku. Kokkutõmbumise kompenseerimiseks võetakse vaiba algpikkus soovitavast lõpp-pikkusest 20% suurem. Samuti arvestatakse, et iga lõime kumbagi otsa tuleb jätta sidumiseks 25 cm varu.



Koostada funktsioon `lõimede_pikkus`, mis

- võtab argumentideks vaiba lõpp-pikkuse (ujukomaarv) ja lõimede arvu (täisarv),
- arvutab ja tagastab vaiba lõimede kogupikkuse ümardatuna sajandikeni.

Vihje

Arvutamiseks võib kasutada valemit: (lõime kogupikkus) = (lõimede arv) * ((vaiba lõpp-pikkus) * 1,2 + 0,5))

Koostada programm, mis

- küsib kasutajalt
  - failinime, kus on vaipade lõpp-pikkused ujukomaarvudena meetrites eraldi ridadel,
  - kõrvuti olevate lõimede arvu 5-meetriste ja pikemate vaipade puhul (täisarv),
  - kõrvuti olevate lõimede arvu lühemate vaipade puhul (täisarv);
- loeb failist vaipade pikkused,
- arvutab (funktsiooni `lõimede_pikkus` abil) ja väljastab ekraanile iga vaiba lõimede kogupikkuse,

- arvutab ja väljastab ekraanile, kui palju läheb lõimeniiti vaja kõigi vaipade peale kokku ümardatuna sajandikeni.

Näide arvutuskäigust

Olgu näiteks uues hoones tahtmine kasutada kaltsuvaipu lõpp-pikkustega 7 m, 4,9 m, 3,63 m ja 5 m. Seejuures 5-meetriste või pikemate lõpp-pikkustega vaipade puhul olgu lõimi kõrvuti 120 (vaiba laius umbes 70 cm), lühemate puhul 140 (vaiba laius umbes 80 cm).

Esimese vaiba üks lõim on pikkusega 7 * 1,2 + 0,5 = 8,9 meetrit. Kokku on sellel vaibal lõimi 120, sest vaiba pikkus on 5 meetrit või rohkem. Seega kulub selle vaiba peale kokku 8,9 * 120 = 1068 meetrit lõimeniiti. Teise, kolmanda ja neljanda vaiba peale kulub vastavalt 893,2; 679,84 ja 780 meetrit. Järelikult on kõikide vaipade jaoks vaja kokku 1068 + 893,2 + 679,84 + 780 = 3421,04 meetrit lõimeniiti.

Näide funktsiooni `lõimede_pikkus` rakendamisest

```
>>> lõimede_pikkus(7, 120)
1068.0
>>>
```

Näide programmi tööst

Faili delta_vaibad.txt sisu:
7
4.9
3.63
5

```
>>> %Run yl8.1a.py
  Sisestage failinimi: delta_vaibad.txt
  Sisestage 5-meetriste ja pikemate vaipade lõimede arv: 120
  Sisestage lühemate vaipade lõimede arv: 140
  1068.0
  893.2
  679.84
  780.0
  Kõigi vaipade peale läheb vaja 3421.04 meetrit lõimeniiti.

>>>
```

Arvestusülesandele sarnane ülesanne koos ühe võimaliku lahendusega võib olla ka abiks mõtete kogumisel. Lahendamise video

KONTROLLÜLESANNE 8.2a ARVESTUSÜLESANDE LAHENDAMISE PROTSESS

Esitamine *Moodle*'is.

Tekstina esitatakse arvamus arvestusülesande raskusastme ja sobivuse kohta. Thonny kasutajad peavad esitama logifaili. Kes teeb mõne teise vahendiga, see peab põhjalikumalt kirjeldama lahendamise raskusi ja kergusi ning samuti peab andma võimalikult täpse ajalise ülevaate, kui palju ülesande lahendamisele aega kulus.

Logifaili saamise video

JUHEND 1 (Thonnyst user_logs'ini)

1. Kui Thonny on avatud, siis pange see kinni (siis tekib viimane logi).
2. Avage Thonny.
3. Valige menüüst Tools.
4. Tools'i alt leiate Open Thonny data folder, mis avab kasutaja Thonny kausta.
5. Avage sellest kaustast user_logs.
6. Sealt valige logid, mille failinimi on seotud antud ülesandega. (Failinimi on kuupäev, mil antud ülesandeid sooritasite.)
7. Tõstke need ühte kausta ning pakkige kokku kas .rar või .zip failiks.
8. Esitage kokkupakitud fail Moodle'i kaudu.

JUHEND 2 (Thonnyst user_logs'ini)

1. Kui Thonny on avatud, siis pange see kinni (siis tekib viimane logi).
2. Avage Thonny.
3. Valige menüüst Tools.
4. Tools'i alt leiate Export usage logs, mis pakib logid zip-failiks kokku.
5. Esitage kokkupakitud fail.

JUHEND 3 (Thonnyt kasutamata user_logs'ini)

1. Kui Thonny on avatud, siis pange see kinni (siis tekib viimane logi).
2. Avage Minu Arvuti/My Computer.
3. Valige sealt kaust, kuhu on installeeritud Windows.
4. Kettalt valige Users kaust, kust leidke kasutaja kaust, kes on kasutanud Thonnyt.
5. Kasutaja kaustas peaks asuma .thonny kaust.
6. .thonny kaust sisaldab kausta User_logs, kuhu on salvestatud Thonny logid.
7. Sealt valige logid, mille failinimi on seotud antud ülesandega. (Failinimi on kuupäev, mil antud ülesandeid sooritasite.)
8. Tõstke need logid ühte kausta ning pakkige kokku kas .rar või .zip failiks.
9. Esitage kokkupakitud fail Moodle' kaudu.

Lähteülesande põhjal on ülesande lahendusel järgmised võimalikud osad:

1) **Funktsioon** `lõimede_pikkus`, mis võtab argumentideks vaiba lõpp-pikkuse (ujukomaarv) ja lõimede arvu (täisarv), arvutab ja tagastab vaiba lõimede kogupikkuse ümardatuna sajandikeni.
2) **Kasutajalt küsimine**. Programm küsib kasutajalt failinime, kus on vaipade lõpp-pikkused ujukomaarvudena meetrites eraldi ridadel, kõrvuti olevate lõimede arvu 5-meetriste ja pikemate vaipade puhul (täisarv), kõrvuti olevate lõimede arvu lühemate vaipade puhul (täisarv).
3) **Failist lugemine.** Programm loeb failist vaipade pikkused. Osa õppijaid koostas tsükli mis loeb failist andmed ja lisab järjendisse. Teine osa kirjutas failist lugemise samasse tsüklisse, mis teeb järgmise osa tegevusi.

4) Iga vaiba lõimede kogupikkuse arvutamine. **Tsükkel ja tingimuslause, funktsiooni kasutamine.** Programm arvutab (funktsiooni lõimede_pikkus abil) ja väljastab ekraanile iga vaiba lõimede kogupikkuse.
5) Lõimeniidi kogupikkuse arvutamine. Programm arvutab ja väljastab ekraanile, kui palju läheb lõimeniiti vaja kõigi vaipade peale kokku ümardatuna sajandikeni.

Lahendus võis olla näiteks selline (joonis 21):

```
def lõimede_pikkus(pikkus, arv):
    return round(arv * (pikkus * 1.2 + 0.5),2)

failinimi = input("Sistage failinimi: ")
lõimede_arv_pikk = int(input("Sistage 5-meetriste ja pikemate vaipade lõimede arv: "))
lõimede_arv_lühike = int(input("Sistage lühemate vaipade lõimede arv: "))

f = open(failinimi)
pikkused = []
for rida in f:
    pikkused.append(float(rida))
f.close()

vaiba_lõimede_pikkus = 0.0
kogupikkus = 0.0
for i in range(len(pikkused)):
    if pikkused[i] >= 5:
        vaiba_lõimede_pikkus = lõimede_pikkus(pikkused[i], lõimede_arv_pikk)
    else:
        vaiba_lõimede_pikkus = lõimede_pikkus(pikkused[i], lõimede_arv_lühike)
    print(str(vaiba_lõimede_pikkus))
    kogupikkus += vaiba_lõimede_pikkus
    kogupikkus = round(kogupikkus,2)

print("Kõigi vaipade peale läheb vaja " + str(kogupikkus) + " meetrit lõimeniiti.")
```

Joonis 21. Näide ülesande lahendusest (lahendaja L11)

3.2.2 Analüüs

Õppijad tegutsevad programmeerimisülesannet lahendades väga mitmekülgselt. Juba päris ülesande lahendamise alguses ilmnesid erinevad käitumismustrid. Enne veel, kui alustati koodi kirjutamist, avas 27 õppijast 15 ühe varasema lahenduse faili, kust vajadusel abi saada. Kuus alustas ilma teisi faile avamata ja kuus avas neid kohe rohkem (üks lahendajatest avas kohe 21 faili). Osa õppijatest vajas aga teiste ülesannete lahenduste näol hiljem lisaabi ja avas neid veel. Selliseid lahendajaid oli 12. Viis lahendajat 27-st vajas ülesande lahendamise jooksul vaatamiseks rohkem kui 10 varasema ülesande lahendust. Kuus lahendajat uuris varasemate ülesannete lahendusi sellisel viisil, et pidas vajalikuks neid ka käivitada.

Suuremate osade kopeerimist teise ülesande lahendusest kasutas kuus õppijat ning neli lahendas ülesannet nii, et jättis varasema ülesande koodi Thonny programmiaknasse ja asus seda modifitseerima ning täiendama. Kaks lahendajat kopeeris programmiaknasse ka ülesande püstituse ja hoidis seda lahendamise ajal väljakommenteerituna silme ees. Suuremate tekstiosade kopeerimise vähene kasutamine erineb Bliksteini (2011, 2013) tulemusest, kus kopeerimine-kleepimine või iseseisvalt kirjutamine sai õppijate gruppidesse jaotamise põhialuseks. Hosseini et al. (2014) aga nägi selge käitumismustrina selleski töös ilmnenud sarnase programmikoodi otsimist ja seejärel täiendamist-parandamist. Ta on sellisel viisil tegutsejaid kirjeldanud lausa alamgrupina.

Suurem osa õppijaid alustas ülesande lahendamist funktsiooni kirjutamisega, kuid leidus ka neid, kes lõid esimesena kasutajalt küsimise osa. 12 õppijat 27-st kirjutas esialgu valmis funktsiooni ja asus siis seda testima. Koguni 11 lahendajat aga käivitas esimest korda alles siis, kui kogu lahenduse esimene versioon oli peaaegu valmis või täiesti valmis. Kui osa jäidki järjest käivitama ja vigu parandama, siis leidus ka neid, kes hakkasid erinevaid programmi osi hiljem eraldi testima. Viis õppijat kasutas programmiosa töö kontrollimiseks eelnevat väljakommenteerimist, viis tegi katsetusi mõnes teises failis ja kolm kustutas ülejäänud koodi ära või kleepis korraks mujale. Kolm õppijat aga reaalselt lahendas ülesannet rohkem kui ühes failis.

Üks õppija lahendas ülesannet nii, et lahenduskäik oli paralleelselt eri versioonides kolmes failis. Ta kirjutas viimasesse faili varasema töötava versiooni ilma kopeerimist kasutamata ümber, mistõttu tekkisid uued vead. Seejärel asus ta taas vigu parandama. Võimalik, et taoline omapärane samm sai ette võetud harjutamiseks. Seitse õppijat 27-st kasutas programmi tööst aru saamiseks silumist. Ühel neist oli selgelt eristuv tööstiil - ta nimelt eelistas programmi töö katsetamiseks peamiselt silumist, kasutades kogu ülesande lahendamise käigus silumist kaheksa korda ja käivitamist vaid kaks korda.

Logisid läbi vaadates jälgiti ka seda, kuivõrd tegid õppijad pärast veateadet midagi muud peale katsete viga parandada. Valdavalt asuti pärast veateadet koodi muutma. 10 õppijal 27-st tuli ette ka seda, et nad käivitasid programmi uuesti, ilma et oleks proovinud vahepeal viga parandada. Samas ei toiminud ükski neist sellisel viisil palju kordi. Üks õppija toimis sama ülesande lahendamisel nii maksimaalselt neli korda. Seega ei olnud selle ülesande puhul korduvate veateadete tulemisel olulist rolli varasemates uuringutes (nt Jadud, 2005) kirjeldatud käitumisviisil, et paljud käivitavad programmi korduvalt uuesti, ilma et prooviks

viga parandada. Ülevaade uuringu aluseks oleva 27 lahenduse korduvatest veateadetest on tabelis 2.

Tabel 2. Korduvad veateated

Jrk	Veateade	Õppijate arv	Korduvate veateadete arv	Maksimaalne arv õppija kohta
1	SyntaxError: invalid syntax	12	98	14
2	NameError: name 'X' is not defined	12	49	5
3	TypeError: '>=' not supported between instances of 'str' and 'int'	8	41	11
4	ValueError: could not convert string to float:	6	38	13
5	TypeError: can't multiply sequence by non-int of type 'float'	5	22	8
6	TypeError: unorderable types: str() >= int()	3	10	5
7	TypeError: unsupported operand type(s) for +=: 'int' and 'tuple'; TypeError: unsupported operand type(s) for +: 'int' and 'str'; TypeError: unsupported operand type(s) for +: 'int' and 'function'	3	9	4
8	TypeError: float() argument must be a string or a number, not 'tuple'; TypeError: int() argument must be a string, a bytes-like object or a number, not 'list'; TypeError: write() argument must be str, not int	3	9	3
9	SyntaxError: unexpected EOF while parsing	3	6	2
10	ValueError: invalid literal for int() with base 10: '\ufeff7\n'; ValueError: invalid literal for int() with base 10: ''	2	10	8
11	TypeError: 'float' object is not iterable	2	7	5
12	SyntaxError: unexpected indent	1	6	6
13	SyntaxError: 'return' outside function	1	3	3
14	SyntaxError: can't assign to comparison	1	3	3
15	TypeError: str() argument 2 must be str, not int	1	3	3
16	AttributeError: 'float' object has no attribute 'round'	1	2	2
17	TypeError: 'bool' object is not callable	1	2	2
18	TypeError: 'float' object is not subscriptable	1	2	2
19	TypeError: bad operand type for unary +: 'str'	1	2	2
20	TypeError: float() takes at most 1 argument (2 given)	1	2	2
21	TypeError: must be str, not float	1	2	2
22	ValueError: I/O operation on closed file	1	2	2

3.3 Õppijatüübid

Kui osa käitumismustritest on juhuslikud või väikese mõjuga, siis leidus ka kriteeriume, mille alusel jaotada õppijad selle uurimuse aluseks oleva MOOC-i kokkuvõtva arvestusülesande lahenduste kontekstis tüüpidesse. Juba läbivaatuse käigus oli näha, et õppijate tegevus erines selle poolest, kas nad töötasid järk-järgult või mitte.

12 õppijat 27-st kirjutas kõigepealt valmis funktsiooni ja siis asus seda testima. Kui funktsioon töötas korrektselt, asuti järgmise osa juurde. Neist omakorda seitse lahendas ka ülejäänud ülesande järk-järgult, kontrollides programmi tööd siis, kui järgmine osa oli kirjutatud. Järgmine käivitus võis olla näiteks siis, kui andmete failist lugemine ja tsükli abil järjendisse lisamine oli tehtud. Ülejäänud viis 12-st olid küll eraldi testinud funktsiooni, kuid järgneva koodi kirjutasid korraga ja hakkasid selle korrektsust testima lõpus. Võimalik, et nad käsitlesidki ülejäänud ülesannet ühe osana. 11 õppijat 27-st kirjutas valmis suure osa või kogu ülesande lahenduse ja asus programmi testima ning parandama pärast seda. Neli õppijat kirjutas valmis osa koodi ja asus seejärel tegema parandusi.

Kuigi varasemad autorid on kasutanud erinevat metoodikat ja võtnud õppijate grupeerimisel arvesse teisi kriteeriume, tuleb järkjärguline tegutsemine vastandatuna muudele käitumismustritele esile rohkem kui ühe autori töös. Näiteks vaatas Blikstein (2011) kirjutatud koodi hulka erinevatel ajahetkedel, kompileerimiste vahelist aega ja veateateid. Gruppide eristajaks saab koodi lisandumine kas astmeliselt (omane peamiselt algajatele, kes kopeerivad-kleebivad rohkem) või ühtlaselt (omane kogenud programmeerijatele).

Õppijad jagas tüüpidesse ka Hosseini et al. (2014), kes võttis gruppidesse jagamise aluseks programmi konstruktsioonide arvu ning kui palju läbis programm teste, arvestades ka kompileerimiste ja salvestamiste arvu. Siin nähakse samuti järkjärgulist tegutsemist (ehitajad, ingl *builders*) ja teisi käitumismustreid (massöörid, ingl *massagers*, kes teevad vahepeal ainult pisimuudatusi), mille hulgas on ka variant, kus õppijad kirjutavad kogu programmi valmis ja hakkavad seejärel selle struktuuri parandama või leiavad sarnase programmikoodi ja hakkavad seda kohendama, lisades vajalikku ja eemaldades ebavajalikku (skulptorid, ingl *reducers*). Viimase variandi puhul võib juhtuda, et testide arv, mida programm läbib, võib töö käigus vahepeal ka kahaneda. Autorid lisasid eraldi gruppi need, kellel läheb kaua aega ja vaeva, et programm hakkaks teste läbima (vaevlejad, ingl *strugglers*). Hosseini et al. (2014) gruppide nimed on õnnestunult eesti keelde tõlkinud Hansalu (2015).

Käesolevas töös koguti kõikide lahenduste kohta mitmekülgset informatsiooni. Järgnevalt vaadati nimetatud järkjärgulist ja mitte järkjärgulist töötamist selle informatsiooni alusel. Andmete kõrvutamine näitas, et järk-järgult töötavad õppijad lahendasid sagedamini ülesande ära kiiremini. Seetõttu kirjeldatakse gruppe eelkõige oluliste tunnuste valguses, mis on seotud lahendamise sujuvusega.

Selleks, et analüüsida õppijate käitumismustreid lahendamise sujuvuse perspektiivis, valiti välja sisuliselt olulised tunnused, mis on peamiselt omased õppijatele, kellel kulub ülesande lahendamiseks rohkem aega. Järgnevalt selgitatakse nende tunnuste valikut.

Põhialuseks võeti korduvad veateated, mille sagedust on peetud üheks selgemaks indikaatoriks, mis näitab, kui hästi õppija õppetöös edasi jõuab. Sellisel seisukohal on näiteks Jadud (2006a). Peatükis 3.2.2 kirjeldatud käitumisviise ajakuluga kõrvutades leiti, et lisaks veateadete ja korduvate veateadete suuremale hulgale, tuli aeglasematel lahendajatel hiljem rohkem uuesti ka varem esinenud veateateid (korduvaks veateateks loeti sama tüüpi veateadet sama koodiosa kohta, süntaksivea puhul vaadati ka fraasi, mis viitab vea asukohale). Samuti hinnati oluliseks näitajaks teiste ülesannete lahenduste kasutamist ülesande lahendamise ajal. Seega vaadati, kui palju oli korduvaid veateateid ning kui palju oli olukorda, kus veateatele järgnes teistsugune veateade ja seejärel tuli varasem uuesti. Samuti vaadati, kui palju avasid õppijad ülesande lahendamise ajal teiste ülesannete lahenduste faile.

Järkjärgulise või mitte-järkjärgulise töötamise kõrvutamisel lahendamise sujuvusega seotud tunnustega kujundati õppijatüübid, millele pandi ka metafoorsed nimed: “müüriladujad” (tabelis 3 violetsega), “kiviraiujad” (tabelis 3 rohelisega) ja “meistrid” (tabelis 3 sinisega). Gruppidest jäeti välja õppijad (neid oli neli), kes ei testinud kõigepealt funktsiooni ega kirjutanud enne testimist valmis ka suuremat osa lahendusest, vaid kirjutasid väiksema osa.

Tabel 3. Töötamise järkjärgulisus ja viis lahendamise sujuvusega seotud olulist tunnust

ID	Kirjutas kõigepealt funktsiooni ja siis testis seda. Jah - 1, Ei - 0	Enne testimist kirjutas valmis suurema osa lahendusest. Jah - 1, Ei - 0	Kestus	Varasemate ülesannete lahenduste arv, mida avas	Veateadete arv	Korduvate veateadete arv	Varasemate veateadete taasesinemisi
L1	1	0	00:15:21	1	1	0	0
L2	1	0	01:35:21	5	26	17	7
L3	0	0	02:49:31	4	23	17	1
L4	0	1	01:45:20	21	35	23	3
L5	1	0	00:59:11	1	11	7	0
L6	1	0	00:16:33	1	6	2	0
L7	0	1	03:39:36	16	39	22	8
L8	0	1	03:12:58	5	51	31	4
L9	0	1	04:34:11	13	55	43	8
L10	1	0	00:57:18	1	14	3	1
L11	1	0	01:14:59	1	4	0	0
L12	1	0	00:35:14	1	13	6	1
L13	0	1	00:41:27	2	10	5	0
L14	1	0	01:00:03	0	16	11	1
L15	0	1	03:55:31	13	36	23	4
L16	0	0	05:47:48	0	48	40	4
L17	0	0	01:15:40	9	11	2	0
L18	1	0	00:30:25	0	7	2	1
L19	0	1	04:06:53	6	41	29	3
L20	1	0	01:28:36	0	7	2	1
L21	1	0	00:26:39	5	4	0	0
L22	0	1	00:39:51	1	20	13	0
L23	0	1	01:00:14	0	3	2	1
L24	0	1	00:33:19	1	2	0	0
L25	0	0	01:50:55	1	6	2	0
L26	1	0	01:04:45	4	11	5	0
L27	0	1	06:35:39	11	32	21	3
	12	**11**		**123**	**532**	**328**	**51**

Järgnevalt selgitatakse, miks valiti õppijatüüpidele just need nimed. Seejärel iseloomustatakse gruppe selle uurimuse aluseks oleva MOOC-i kokkuvõtva arvestusülesande lahenduste kontekstis.

1) “Müüriladujad”. Selle grupi õppija töötab järk-järgult. Ta loob kõigepealt ühe osa programmist ja testib seda, siis alles läheb järgmise osa juurde. Seda saab võrrelda müüriladuja tööga, kuna töö tulemusel kerkib müür järk-järgult. Ta laob eelmise telliserea ära ja alles siis alustab järgmisega.

2) “Kiviraiujad”. Selle grupi õppija teeb kõigepealt esialgse versiooni programmist valmis ja hakkab seda siis lihvima, selleks läheb palju vaeva ja abivahendina on vaja palju teisi faile. Seda saab võrrelda kiviraiuja tööga, kus on kõigepealt vaja algmaterjali, millest hakata sobivat eset raiuma. See on raske töö, mis nõuab palju vaeva. Samuti on vaja palju erinevaid tööriistu.

3) “Meistrid”. Selle grupi õppija teeb samuti kõigepealt esialgse versiooni programmist valmis, aga see tuleb tal hästi välja ja veateateid on vähe või suudab õppija vead kiiresti parandada. Teda võib võrrelda meistriga, kes on osav ja teab, mida teeb. Kui vaadata järkjärgulisust, siis meenutab tegevus "kiviraiujat" - lahendus tehakse kohe valmis. Tegelikult aga võib see olla ennatlik klassifitseerimine, sest ülesanne võib tema jaoks olla lihtsalt piisavalt lihtne ühes järgus lahendamiseks.

Selle uurimuse aluseks oleva MOOC-i kokkuvõtva arvestusülesande lahenduste kontekstis saab õppijatüüpe iseloomustada järgmiselt:

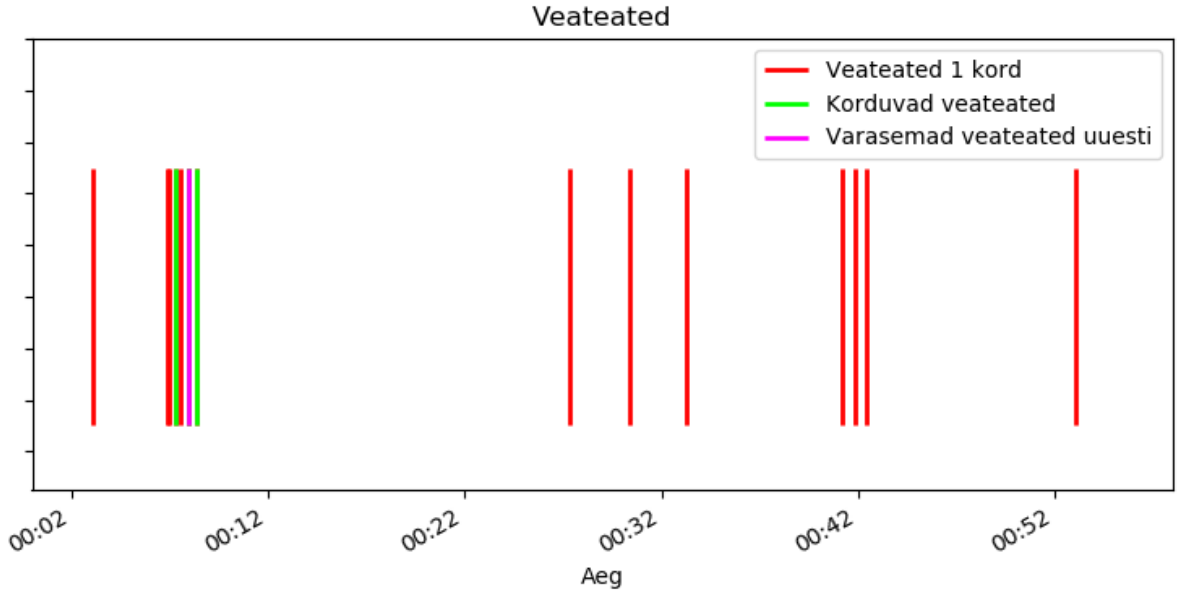
1) “Müüriladujad”

Tabelis 3 lahendajad L1, L2, L5, L6, L10, L11, L12, L14, L18, L20, L21 ja L26.

Nad lisasid koodi järk-järgult ja käivitasid esimest korda juba siis, kui üsna väike osa programmist oli valmis - funktsioon. Vigu parandasid nad järk-järgult ehitamise käigus ja kasutasid varasemate ülesannete tuge vähe. Neil oli ülesande lahendamise käigus ülevaade, kas juba loodud osad töötasid õigesti või mitte. “Müüriladujate” hulgas on võimalik eristada kahte alamgruppi. Esimene grupp neist testis programmi korduvalt (mitte ainult pärast funktsiooni ja päris lõpus), teine grupp testis programmi tööd pärast funktsiooni kirjutamist, kuid ülejäänu kirjutas korraga ning testis uuesti alles lõpus. Suure tõenäosusega ei

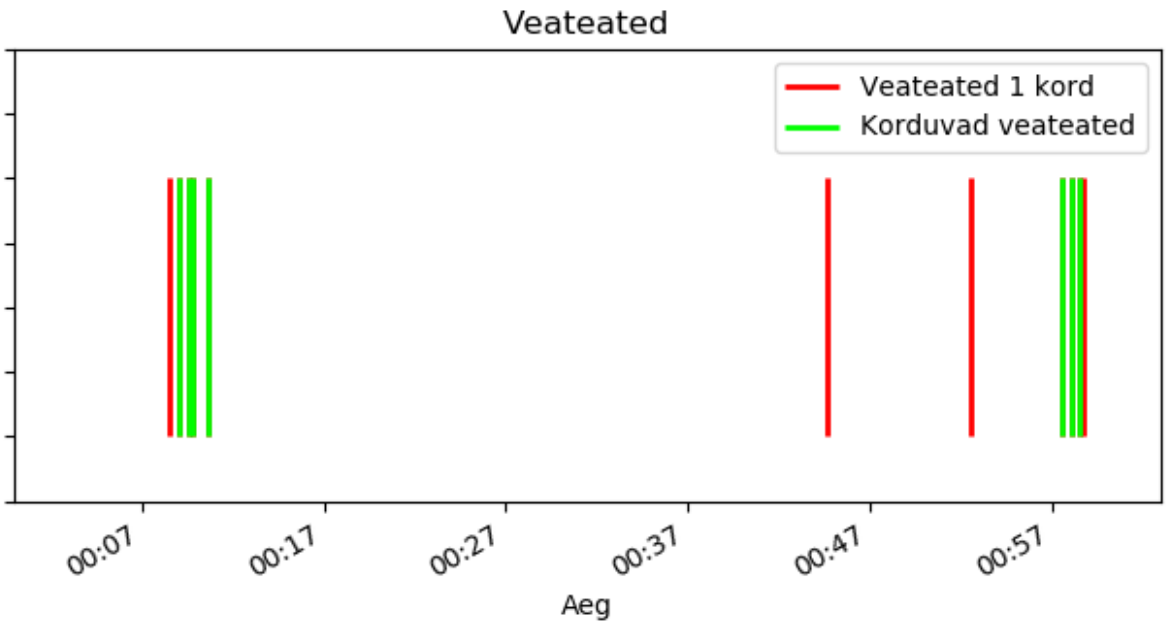
valmistanud see ülesanne keerukust ja seetõttu võis pärast funktsiooni testimist teha pika sammu ehk lahendada ülejäänud osa korraga.

Seitsmest "müüriladujast", kes testis programmi ka vahepeal (mitte ainult pärast funktsiooni ja päris lõpus), ei avanud kaks ühtegi lisafaili, kolm avas ühe. Vaid kaks nende hulgast vajas rohkem teiste ülesannete lahenduste faile: üks neist avas viis faili ja teine neli. Ajakulu oli neil kõigil "kiviraiujatega" võrreldes väiksem - ka neil kahel, kes avas rohkem faile. Üldiselt oli "müüriladujatel" vähe korduvaid veateateid. Kuuel seitsmest näiteks 5, 3, 2, 2, 0 ja 0 (võrdluseks: kõigil "kiviraiujate" gruppi kuulujatel oli korduvaid veateateid üle 20). Ühel aga oli 17 korduvat veateadet. Olukorda, kus veateatele järgnes teistsugune veateade ja seejärel tuli varasem uuesti, ei olnud kolmel seitsmest ühtegi, kolmel oli üks kord ja ühel seitse (vt selle grupi näidet veateadete paiknemisest ajateljel jooniselt 22).



Joonis 22. Ühe "müüriladuja" veateadete paiknemine ajateljel (lahendaja L10)

"Müüriladujad", kes testisid programmi pärast funktsiooni ja seejärel päris lõpus (neid oli kokku viis) ei vajanud ükski töö käigus rohkem kui ühte varasema ülesande lahendust ja keegi ei lahendanud ülesannet üle tunni (võrdluseks: "kiviraiujate" grupis vajasid kõik peale ühe aega üle kolme tunni ja kolm koguni üle nelja tunni). Korduvaid veateateid oli ka oluliselt vähem. Olukorda, kus veateatele järgnes teistsugune veateade ja seejärel tuli varasem veateade uuesti, ei olnud kolmel ühtegi ja ülejäänud kahel üks kord (vt selle grupi näidet veateadete paiknemisest ajateljel jooniselt 23).

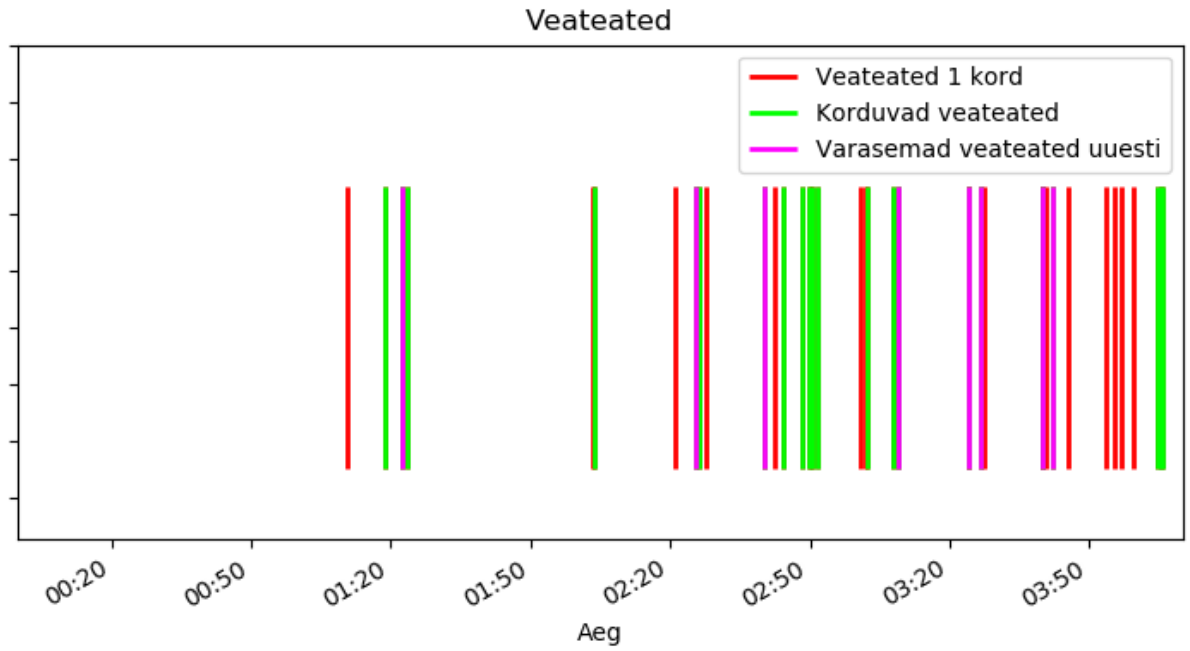


Joonis 23. Ühe “müüriladuja” veateadete paiknemine ajateljel (lahendaja L5)

2) “Kiviraiujad”

Tabelis 3 lahendajad L4, L7, L8, L9, L15, L19 ja L27.

Nad kirjutasid valmis suurema osa ülesande lahendusest ja asusid alles siis programmi tööd testima ja koodi parandama. Kirjutamine aga ei tahtnud väga ladusalt sujuda, seega vajasid nad ülesande lahendamise ajal paljude varasemate ülesannete lahenduste faile. Näiteks vajasid seitse “kiviraiuja” vaatamiseks vastavalt 21, 16, 13, 13, 11, 6 ja 5 varasema ülesande lahendust. Neil tuli lahendamise käigus palju korduvaid veateateid (koguarvud suuruse järjekorras 43, 31, 29, 23, 23, 22 ja 21), mille hulgas oli kõigil ka selliseid, kus veateatele järgnes teistsugune veateade ja seejärel tuli varasem uuesti (8, 8, 4, 4, 3, 3, 3) (vt selle grupi näidet veateadete paiknemisest ajateljel jooniselt 24). Neil läks lahendamiseks ka teistest oluliselt rohkem aega. Kõik kauem ülesannet lahendanud kirjutasid enne esimest käivitamist suurema osa lahendusest valmis. Pärast esimese tervikversiooni käivitamist kujunes põhitegevuseks leida korduvate käivitamiste ja parandamiste järel üles kõik vead, mis programmis olid.

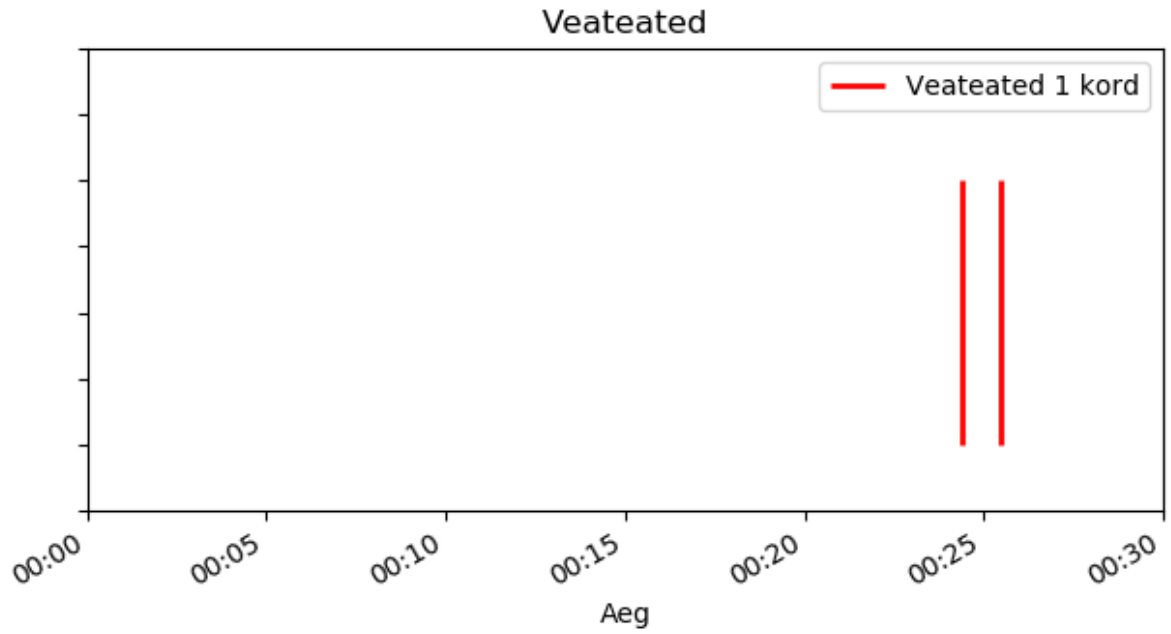


Joonis 24. Ühe “kiviraiuja” veateadete paiknemine ajateljel (lahendaja L7)

3) “Meistrid”

Tabelis 3 lahendajad L13, L 22, L23 ja L24.

Nad kirjutasid valmis kogu lahenduse ja alles siis käivitasid esimest korda, aga kuna programmis oli vähe vigu või suutis õppija vead kiiresti parandada, siis kõik sujus. Teisi faile vajasid nad abivahendina vähe. Neljast “meistrist” üks avas kaks faili, järgmised kaks ühe faili ja neljas mitte ühtegi. Samuti läks neil “kiviraiujatest” oluliselt vähem aega ja oli vähem veateateid. Nende lahendamise sujuvus on pigem sarnane “müüriladujatega”. Ühel oli veateateid rohkem (20), aga ta parandas need kiiresti. Teine sai kogu lahendamise jooksul 10 veateadet, kolmas kolm ja neljas vaid kaks (vt selle grupi näidet veateadete paiknemisest ajateljel jooniselt 25).



Joonis 25. Ühe “meistri” veateadete paiknemine ajateljel (lahendaja L24)

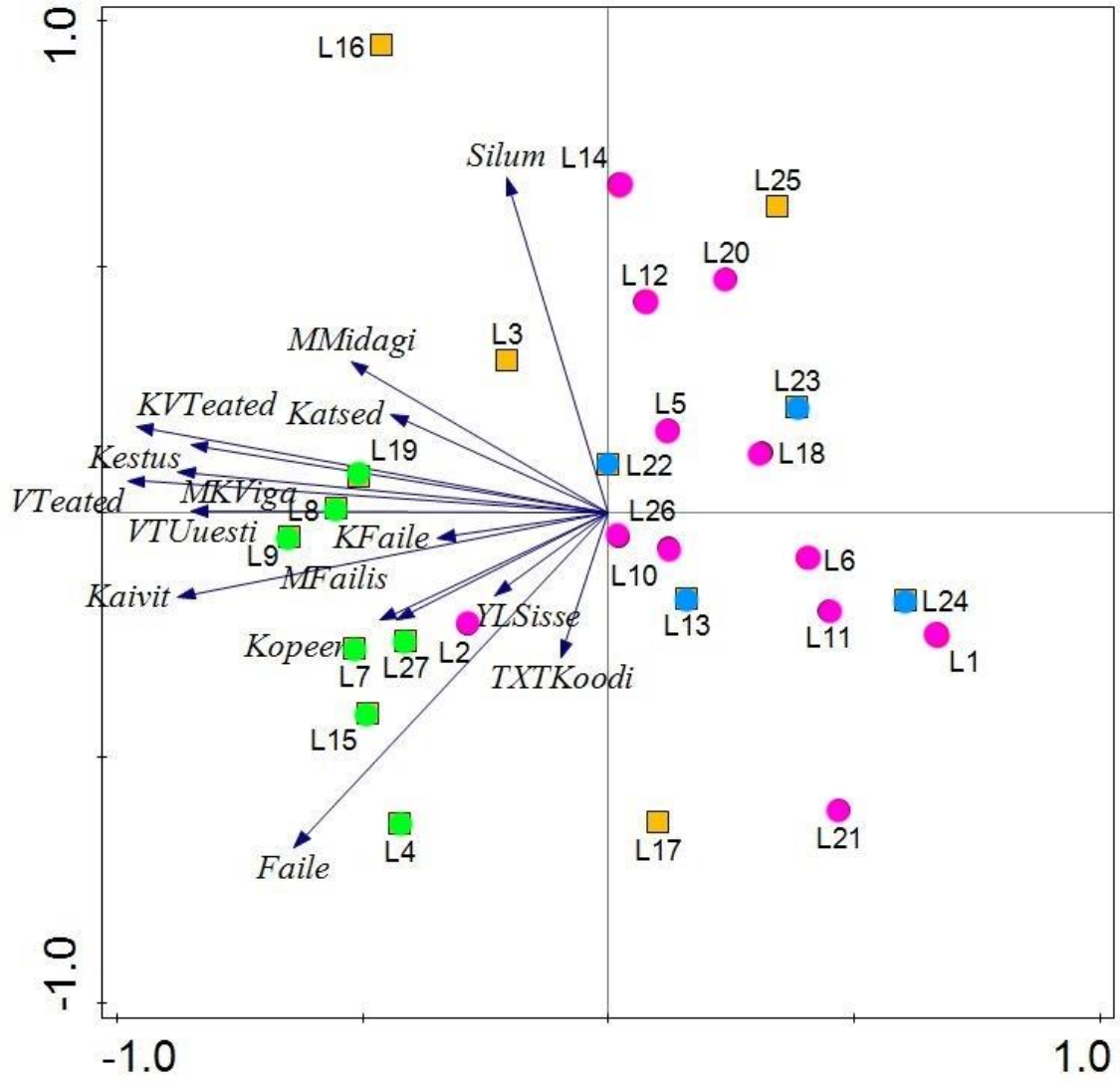
Kui gruppe võrrelda, siis kõik 12 "müüriladujat", kes kirjutasid kõigepealt valmis funktsiooni ja asusid seda siis testima, lahendasid ülesande kiiremini kui ükski seitsmest "kiviraiujast", kes kirjutas enne testimist valmis suurema osa ülesande lahendusest. Kõigil seitsmel “kiviraiujal” oli ka rohkem veateateid ja korduvaid veateateid kui ühelgi “müüriladujal”. Võib öelda, et seitsmest ülesannet kõige vähem sujuvalt lahendanud õppijast ei töötanud keegi järk-järgult. "Meistrite" grupi viis liiget kirjutasid enne testimist valmis suurema osa ülesande lahendusest, kuid said sellega hästi hakkama. Ei ole teada, kas "meistrid" töötavad järk-järgult ja analüüsi aluseks olnud MOOC-i kokkuvõttev arvestusülesanne oli nende jaoks sedavõrd lihtne, et nad käsitlesid seda ühe osana, või on olemas õppijate grupp, kelle puhul mitte-järkjärguline käitumismuster toimib. Edasised uuringud suurema valimiga saavad anda täpsustavat teavet.

Hosseini et al. (2014) liigitus on küll teistel alustel (vaatab programmi konstruktsioonide arvu ja testide läbimist käivitamiste ja salvestamiste ajal), kuid ka seal on välja toodud õppijad, kes töötavad järk-järgult, ja õppijad, kes kirjutavad kohe kogu programmi valmis ja asuvad siis selle struktuuri muutma. Seal tähendas valmis kirjutamine ka testide läbimist. Need, kellel läheb kaua aega ja vaeva, et programm hakkaks teste läbima, on autorid lisanud veel omaette gruppi.

Järgnevalt otsiti võimalusi gruppide paiknemist visualiseerida. Samuti püüti leida viise, kuidas kaardistada tunnuste omavahelisi seoseid. Konsulteerides taimeökoloogiga leiti, et üks võimalik viis gruppide visualiseerimiseks ja tunnuste omavaheliste seoste esmaseks kaardistamiseks praeguse valimi suuruse juures võiks olla peamiselt ökoloogias kasutatav ordinatsioonanalüüs (ingl *ordination analysis*), mille kohta on Remm, Remm ja Kaasik (2012) kirjutanud: "Mitmemõõtmeline ordinatsioon püüab analoogiliselt faktoranalüüsiga kirjeldada vaatluste või tunnuste paiknemist tunnusruumis võimalikult väheste telgede abil. Erinevalt faktor- ja peakomponentanalüüsist ei nõua mitmemõõtmeline ordinatsioon tunnuste normaaljaotust. Ordineerida saab igasuguseid tunnuseid ja vaatlusi, mille puhul on võimalik arvutada nende omavahelist sarnasust." Ordinatsioonanalüüsi on programmeerimise õppimise uurimisel kasutanud näiteks Mather (2015).

Kasutati programmi CANOCO 5.0. Alguses mõõdeti tunnuste koguvarieeruvust CA (ingl *correspondence analysis*) meetodiga, et testida PCA (ingl *principal component analysis*) meetodi rakendatavust. Tulemus 0,16 oli alla 2, seega sai rakendada PCA meetodit tulemuse analüüsiks. Andmed, mille põhjal analüüs tehti, on tabelis 4.

Joonis 26 illustreerib nii tunnuste omavahelisi seoseid kui ka lahendajate paiknemist selles tunnusruumis. Mida lähemal on tunnuseid tähistavad nooled omavahel, seda tugevam on nendevaheline positiivne korrelatsioon (näiteks korreleeruvad tugevalt veateated ja kestus). Mida pikem on noolega joon, seda tugevam on tunnuse mõju. Kuna nooled näitavad kõik sarnases suunas, siis saab joonise 26 põhjal öelda: mida rohkem jääb õppija ikoon vertikaalsest teljest vasakule, seda rohkem läks tal ülesande lahendamiseks aega, seda rohkem oli tal veateateid, korduvaid veateateid jne. Ringiga on tähistatud õppijad, kes kirjutasid kõigepealt valmis funktsiooni ja asusid seda testima, ning ruuduga õppijad, kes alustasid testimisega hiljem.



Müüriladujad Kiviraiujad Meistrid Ei kuulu gruppi

Kestus – kestus; **YLSisse** - kirjutab töö teise ülesande lahenduse sisse; **Kopeer** - kopeerib teise ülesande lahendusest; **TXTKoodi** - kopeerib ülesande teksti väljakommenteerituna koodi; **Faile** - avab varasemate ülesannete lahenduste faile; **KFaile** - käivitab varasemate ülesannete lahenduste faile; **Katsed** - testib koodiosa teistes failides; **MFailis** - lahendab ülesannet mitmes failis; **VTeated** - veateated; **Kaivit** - käivitamised; **Silum** - silumised; **KVTeated** - korduvad veateated; **MKViga** - erinevad korduvad veateated; **VTUuesti** - varasem veateade tuleb hiljem uuesti; **MMidagi** - ei tee pärast veateadet mitte midagi.

Joonis 26. Lahendajate paiknemine paljumõõtmelises tunnusruumis koos gruppidesse jagunemisega

Joonis 26 on kooskõlas tulemustega, milleni jõuti andmete analüüsimisel tabelite põhjal. Paistab kinnitust leidvat, et "müüriladujad" ja "kiviraiujad" eristuvad rühmadena. "Meistrid" tunduvad pigem olevat sarnased "müüriladujatega", kuid oletus vajab täpsemat uurimist

suurema valimi põhjal. Kui tulevikus uurida suuremat hulka õppijaid, saab kasutada erinevaid meetodeid nii gruppide analüüsimisel kui ka tunnuste omavaheliste seoste leidmiseks.

Tabel 4. Andmete tabel

	Kestus	YLSisse	Kopeer	TXTKoodi	Faile	KFaile	Katsed	MFailis	VTeated	Kaivit	Silum	KVTeated	MKViga	VTUuesti	MMidagi
L1	15.35	0	0	0	1	0	0	0	1	6	0	0	0	0	0
L2	95.35	0	0	0	5	0	0	0	26	54	0	17	3	7	0
L5	59.18	0	0	0	1	0	0	0	11	14	0	7	2	0	2
L6	16.55	0	0	0	1	0	0	0	6	9	0	2	1	0	0
L10	57.30	0	0	0	1	0	0	0	14	39	0	3	1	1	0
L11	74.98	0	0	0	1	0	0	0	4	17	0	0	0	0	0
L12	35.23	0	0	0	1	1	0	0	13	23	4	6	2	1	1
L14	60.05	0	0	0	0	0	0	0	16	22	3	11	3	1	0
L18	30.42	0	0	0	0	0	0	0	7	19	0	2	1	1	0
L20	88.60	0	0	0	0	0	1	0	7	16	2	2	1	1	0
L21	26.65	0	0	0	5	0	0	0	4	13	0	0	0	0	0
L26	64.75	1	1	0	4	1	0	0	11	44	3	5	2	0	0
L3	169.52	0	0	0	4	0	0	0	23	56	4	17	0	1	4
L4	105.33	0	0	1	21	0	1	1	35	63	0	23	6	3	0
L7	219.60	0	1	0	16	0	0	0	39	72	1	22	4	8	2
L8	192.97	1	1	0	5	1	0	3	51	95	1	31	10	4	1
L9	274.18	0	1	0	13	1	1	0	55	59	2	43	13	8	0
L13	41.45	1	1	1	2	0	0	0	10	17	0	5	2	0	0
L15	235.52	1	1	0	13	0	0	1	36	59	0	23	7	4	1
L16	347.80	0	0	0	0	0	1	0	48	56	2	40	7	4	3
L17	75.67	0	0	0	9	1	0	0	11	38	0	2	1	0	0
L19	246.88	0	0	0	6	1	1	0	41	78	1	29	9	3	1
L22	39.85	0	0	0	1	0	0	0	20	30	0	13	4	0	0
L23	60.23	0	0	0	0	0	0	0	3	7	0	2	1	1	0
L24	33.32	0	0	0	1	0	0	0	2	5	0	0	0	0	0
L25	110.92	0	0	0	1	0	0	0	6	2	8	2	1	0	0
L27	395.65	0	0	0	11	0	0	0	32	39	0	21	4	3	1

Kestus - kestus minutites; **YLSisse** - kirjutab töö teise ülesande lahenduse sisse; **Kopeer** - kopeerib teise ülesande lahendusest; **TXTKoodi** - kopeerib ülesande teksti väljakommenteerituna koodi; **Faile** - avab varasemate ülesannete lahenduste faile; **KFaile** - käivitab varasemate ülesannete lahenduste faile; **Katsed** - testib koodiosa teistes failides; **MFailis** - lahendab ülesannet mitmes failis; **VTeated** - veateated; **Kaivit** - käivitamised; **Silum** - silumised; **KVTeated** - korduvad veateated; **MKViga** - erinevad korduvad veateated; **VTUuesti** - varasem veateade tuleb hiljem uuesti; **MMidagi** - ei tee pärast veateadet mitte midagi.

Kui uurida suuremat rühma õppijaid, siis väärivad eraldi tähelepanu õppijad, kelle ülesande lahendamise sujuvus on sarnane “kiviraiujatega”, aga kes on omaks võtnud järkjärgulise töötamise. Selle töö valimisse sattus selliseid vaid üks: avas viis faili, korduvaid veateateid oli 17, olukorda, kus veateatele järgnes teistsugune veateade ja seejärel tuli varasem uuesti, oli seitse korda. Tal läks võrreldes “kiviraiujatega” ülesande lahendamiseks vähem aega.

Võimalik, et järkjärgulise töötamise soovitamine võib aidata algajatel programmi tööst paremini aru saada ja vähendada töö käigus olukordi, kus koodis on korraga palju vigu ja lõpuks on raske mõista, mis nüansi tõttu veateade tuli. Kuna õppijatel oli kohati raskusi funktsiooni väljakutsumisega, vajab ka see eraldi tähelepanu.

Jäid silma ka mõned veateadetega töötamise viisid, millest võiks pigem hoiduda:

1) Osa õppijaid teeb pärast veateadet nii mitmes kohas erinevaid parandusi ja täiendusi, et lõpuks ei saa enam aru, milline viga sai parandatud ja milline juurde tehtud.

2) Osa õppijaid ei käivita hetkel, mil oleks oluline veenduda, kas programm töötab. Näiteks üks lahendaja maadles tükk aega veaga ja lõpuks sai programmi tööle. Seejärel tegi samalaadse paranduse teises kohas ja enam ei käivitanud. Kui uus veateade tuli, siis ta taas ei teadnud, kuidas seda parandada.

3.4 Võimalikud edasised uuringud

Kvalitatiivne väikese arvu õppijate ja ühe MOOC-i kokkuvõtva arvestusülesande põhjal tehtud uuring on näidanud mitmeid võimalikke seoseid ja käitumismustreid. Vajalik on uurida õppijate käitumist suurema arvu lahendajate ja paljude erinevate ülesannete põhjal, kasutades ka kvantitatiivseid meetodeid. Samuti on oluline kaardistada õppijate käitumismustreid, kui nad lahendavad ülesandeid aja ja koha piiranguga tingimustes. Eraldi vajab tähelepanu, kas Thonny võimaluste põhjalikum tutvustamine ja tähelepanu järkjärgulisele töötamisele parandab õppijate tulemusi ning aitab vigadest paremini aru saada. Kui uurida suuremat rühma õppijaid, võib tulemusi anda nende õppijate tegevuse analüüsimine, kelle lahendamise sujuvus on sarnane “kiviraiujatega”, aga kes on omaks võtnud järkjärgulise töötamise. Oluline on välja töötada ka võimalused, kuidas õppija saaks logiandmete põhjal pärast ülesande lahendamist kasulikku tagasisidet.

Paljude õppijate logifailide üheaegseks analüüsimiseks ning konkreetsele uurimisfookusele keskendumiseks oleks kasulik luua täiendavaid töövahendeid ja arendada olemasolevaid.

Kokkuvõte

Programmeerimise õppimisel on suur osa iseseisval ülesannete lahendamisel ja õpetajad näevad, millise tulemuseni on õppijad jõudnud, kuid enamasti ei saa selget ja süstemaatilist ülevaadet ülesannete lahendamise protsessi kohta. Tartu Ülikooli õppejõu Aivar Annamaa väljatöötatud Pythoni programmeerimiskeskkond Thonny talletab programmeerimise käigus logifaili samm-sammult detailset informatsiooni kasutaja tegevuse kohta. Seega on logifailide näol olemas väärtuslikud algandmed, mida on võimalik analüüsida ja nii saada õppeprotsessi kohta teada mitmekülgset informatsiooni.

Magistritöös seati eesmärgiks logifailide põhjal leida vastused järgmistele küsimustele:

1) Millist informatsiooni on võimalik Thonny logifailidest koguda õppijate tegevuse kohta?
2) Millised on õppijate erinevad käitumismustrid programmeerimisülesannete lahendamisel MOOC-i "Programmeerimise alused" kokkuvõtva arvestusülesande näitel?
3) Milliseid õppijatüüpe saab eristada ülesande lahendamise käitumismustrite alusel?

Õppeprotsessi uurimiseks kasutati Tartu Ülikooli 2017. aasta sügis-talvel toimunud vaba ligipääsuga e-kursuse "Programmeerimise alused" arvestusülesande logisid. Esmalt analüüsiti põhjalikult logifailide ülesehitust ja seal sisalduvat informatsiooni. See analüüs oli hiljem aluseks ka järgmiste sammude planeerimisel ja logidest vajalikku informatsiooni koguva programmi koostamisel. Lisaks vaadati logid Thonny kasutussessiooni taasesitamise funktsionaalsust kasutades läbi, et märgata õppijate üldisemaid käitumismustreid. Magistritööga seoses kirjutati kaks programmi: valitud andmeid logist tabelisse kirjutav programm ja graafikuid koostav programm.

1) Millist informatsiooni on võimalik Thonny logifailidest koguda õppijate tegevuse kohta?

Logifailid sisaldavad kirjeid paljude eri tüüpi sündmuste kohta, näiteks teksti sisestamised, kustutamised, kopeerimised, kleepimised, faili avamised, programmi salvestamised, käivitamised, silumised ja veateadete kuvamised. Seda infot on võimalik masinlikult koguda. Kuna kõikides kirjetes on olemas võti-väärtus paar tegevuse ajatempli kohta, siis on võimalik arvutada ka ajavahemikke erinevate sündmuste vahel. Masinlikult on võimalik leida ka tähemärkide arvu sündmuste (näiteks käivitamiste) hetkel. Veateadete puhul on näiteks olemas ka tekstiline viide vea täpsemale asukohale, samuti rea number, kus viga asub.

Logifailidest info kogumisel tuleb arvestada, et logid sisaldavad detailset informatsiooni ja ühe sündmuse kohta võib olla mitmeid kirjeid. Ühes logifailis võib olla mitme ülesande lahendus ja ühe ülesande lahendus võib olla mitmes logifailis. Samuti avavad õppijad lahendamise käigus teiste ülesannete lahenduste faile ning aeg-ajalt ka käivitavad neid. Selleks, et eristada logi põhjal üksteisest ühe lahendaja tegevusi mitme Pythoni failiga, on vaja kasutada võtit "text_widget_id". Kui luua programme, mis kasutavad logifailide informatsiooni, siis tuleb arvestada ka sellega, et võti-väärtus paaride paigutus kirjes võib olla erinev sõltuvalt Thonny versioonist, mida õppija on kasutanud.

Kuna logifaile on võimalik kasutussessiooni taasesitamise funktsionaalsust kasutades läbi vaadata ja näha algusest lõpuni, kuidas õppija ülesannet lahendas, siis saab logidest ka palju sellist väärtuslikku teavet, mida masinlikult ei ole võimalik koguda. Samuti andis läbivaatus võimaluse märgata käitumismustreid, mille olemasolu ei olnud ette teada. Näiteks vaadati, kas õppija testis funktsiooni eraldi või mitte, kas ta töötas järk-järgult või mitte, kuivõrd ta avas ülesande lahendamisel teiste ülesannete lahenduste faile või käivitas neid, kas õppija lahendas ülesannet varasema ülesande lahenduse koodi kohendades, kas ta lahendas ülesannet ühes või mitmes failis ja mida tegi pärast veateadet.

2) Millised on õppijate erinevad käitumismustrid programmeerimisülesannete lahendamisel MOOC-i "Programmeerimise alused" kokkuvõtva arvestusülesande näitel?

Logifailidest kogutud mitmekülgne info näitas, et õppijad käitusid MOOC-i kokkuvõtva arvestusülesande lahendamisel erinevalt. Osa õppijaid avas ülesande lahendamise jooksul paljude varasemate ülesannete faile, kust vajadusel abi saada. Oli ka lahendajaid, kes uurisid varasemaid ülesandeid sellisel viisil, et pidasid vajalikuks neid ka käivitada. Suuremate koodiosade kopeerimist teisest ülesandest kasutati samuti, kuid küllaltki vähe. See erineb Bliksteini (2011, 2013) tulemusest, kus kopeerimine-kleepimine või iseseisvalt kirjutamine sai õppijate gruppidesse jaotamise põhialuseks.

Osa õppijaid lahendas ülesannet nii, et jättis varasema ülesande koodi Thonny programmiaknasse ja asus seda modifitseerima ning täiendama. Programmi töö testimiseks kasutati nii eelnevat väljakommenteerimist, katsetusi mõnes teises failis kui ka ülejäänud koodi kustutamist. Mõned õppijad lahendasid ülesannet rohkem kui ühes failis.

Pärast veateadet asuti valdavalt koodi muutma. Õppijatel tuli ette ka seda, et nad käivitasid programmi uuesti, ilma et oleks proovinud vahepeal viga parandada. Samas ei toiminud ükski

neist sellisel viisil palju kordi. Seega ei olnud selle ülesande puhul korduvate veateadete tulemisel olulist rolli varasemates uuringutes (nt Jadud, 2005) kirjeldatud käitumisviisil, et paljud käivitavad programmi korduvalt uuesti, ilma et prooviks viga parandada.

3) Milliseid õppijatüüpe saab eristada ülesande lahendamise käitumismustrite alusel?

Analüüsi tulemusel eristati selle uurimuse aluseks oleva MOOC-i kokkuvõtva arvestusülesande lahenduste kontekstis järgmisi õppijatüüpe:

1) "Müüriladujad", kes lisasid koodi järk-järgult ja käivitasid esimest korda juba siis, kui üsna väike osa programmist oli valmis - funktsioon. Vigu parandasid nad järk-järgult ehitamise käigus ja kasutasid varasemate ülesannete lahenduste tuge vähe. Neil oli ülesande lahendamise käigus ülevaade, kas juba loodud osad töötasid õigesti või mitte. Neil läks "kiviraiujatest" vähem aega ning oli vähem veateateid ja korduvaid veateateid.
2) "Kiviraiujad", kes kirjutasid valmis suurema osa ülesande lahendusest ja asusid alles siis programmi tööd testima ja koodi parandama. Kirjutamine aga ei tahtnud väga ladusalt sujuda, seega vajasid nad ülesande lahendamise ajal paljude varasemate ülesannete lahenduste faile. Neil läks lahendamiseks teiste gruppidega võrreldes rohkem aega ja tuli palju korduvaid veateateid. Pärast esimese tervikversiooni käivitamist kujunes põhitegevuseks leida korduvate käivitamiste ja parandamiste järel üles kõik vead, mis programmis olid.
3) "Meistrid", kes kirjutasid valmis suurema osa ülesande lahendusest ja alles siis käivitasid esimest korda, aga kuna programmis oli vähe vigu või suutis õppija vead kiiresti parandada, siis kõik sujus. Neil läks "kiviraiujatega" võrreldes vähem aega ja nad vajasid teiste ülesannete lahenduste faile abivahendina vähem. Nende lahendamise sujuvus oli pigem sarnane "müüriladujatega".

Kui gruppe võrrelda, siis kõik 12 "müüriladujat", kes kirjutasid kõigepealt valmis funktsiooni ja asusid seda siis testima, lahendasid ülesande kiiremini kui ükski seitsmest "kiviraiujast", kes kirjutas enne testimist valmis suurema osa ülesande lahendusest. Kõigil seitsmel "kiviraiujal" oli ka rohkem veateateid ja korduvaid veateateid kui ühelgi "müüriladujal". Võib öelda, et seitsmest ülesannet kõige vähem sujuvalt lahendanud õppijast ei töötanud keegi järk-järgult. "Meistrite" grupi viis liiget kirjutasid enne testimist valmis suurema osa ülesande lahendusest, kuid said sellega hästi hakkama. Ei ole teada, kas "meistrid" töötavad järk-järgult, kuid analüüsi aluseks olnud MOOC-i kokkuvõttev arvestusülesanne oli nende

jaoks sedavõrd lihtne, et nad käsitlesid seda ühe osana, või on olemas õppijate grupp, kelle puhul mitte-järkjärguline käitumismuster toimib. Edasised uuringud suurema valimiga saavad anda täpsustavat teavet.

Võimalik, et järkjärgulise töötamise soovitamine võib aidata algajatel programmi tööst paremini aru saada ja vähendada töö käigus olukordi, kus koodis on korraga palju vigu ja lõpuks on raske mõista, mille tõttu veateade tuli. Kuna õppijatel oli kohati raskusi funktsiooni väljakutsumisega, vajab ka see eraldi tähelepanu.

Kirjandus

Ahadi, A., Lister, R., Lal, S. & Hellas, A. (2018). Learning programming, syntax errors and institution-specific factors. In *Proceedings of the 20th Australasian Computing Education Conference* (pp. 90-96). ACM. https://dl.acm.org/citation.cfm?id=3160490 (13.03.2018)

Allevato, A. & Edwards, S. H. (2010). Discovering patterns in student activity on programming assignments. In *2010 ASEE Southeastern Section Annual Conference and Meeting*. http://www.softwareeducationsupport.com/ASEE%20SE%20Conference%20Proceedings/Conference%20Files/ASEE2010/Papers/PR2010All158.PDF (19.05.2018)

Altadmri, A. & Brown, N. C. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 522-527). ACM. http://www.twistedsquare.com/37Million.pdf (19.05.2018)

Annamaa, A. (2015a). Thonny: a Python IDE for Learning Programming. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 343-343). ACM. https://dl.acm.org/citation.cfm?id=2754849 (15.02.2018)

Annamaa, A. (2015b). Introducing Thonny, a Python IDE for learning programming. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research* (pp. 117-121). ACM. https://dl.acm.org/citation.cfm?id=2828969 (15.02.2018)

Annamaa, A., Hansalu, A. & Tõnisson, E. (2015). Automatic analysis of students' solving process in programming exercises. In *IFIP TC3 Working Conference "A New Culture of Learning: Computing and next Generations"* (p. 11). (15.02.2018)

Annamaa, A. (s.a.). *TÜ Arvutiteaduse instituudi programmeerimise algkursuse õpik*. http://progeopik.cs.ut.ee/ (22.03.2018)

Aramaa, A. (2014). *Pythoni koodi muudatuste analüsaator* (Bakalaureusetöö). Tartu Ülikool, arvutiteaduse instituut. https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=40741&year=2014 (15.03.2018)

Balzuweit, E. & Spacco, J. (2013). SnapViz: visualizing programming assignment snapshots. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education* (pp. 350-350). ACM. https://dl.acm.org/citation.cfm?id=2465615 (10.03.2018)

Becker, B. A. (2016a). An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (pp. 126-131). ACM. https://dl.acm.org/citation.cfm?id=2844584 (1.03.2018)

Becker, B. A. (2016b). A new metric to quantify repeated compiler errors for novice programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 296-301). ACM. https://dl.acm.org/citation.cfm?id=2899463 (1.03.2018)

Blikstein, P. (2011). Using learning analytics to assess students' behavior in open-ended programming tasks. In *Proceedings of the 1st international conference on learning analytics and knowledge* (pp. 110-116). ACM. https://dl.acm.org/citation.cfm?id=2090132 (19.05.2018)

Blikstein, P. (2013). Multimodal learning analytics. In *Proceedings of the third international conference on learning analytics and knowledge* (pp. 102-106). ACM. https://dl.acm.org/citation.cfm?id=2460316 (19.05.2018)

Denny, P., Luxton-Reilly, A., Tempero, E. & Hendrickx, J. (2011). Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education* (pp. 208-212). ACM. https://dl.acm.org/citation.cfm?id=1999807 (1.03.2018)

Denny, P., Luxton-Reilly, A. & Tempero, E. (2012). All syntax errors are not equal. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education* (pp. 75-80). ACM. https://dl.acm.org/citation.cfm?id=2325318 (1.03.2018)

Denny, P., Luxton-Reilly, A., & Carpenter, D. (2014). Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 conference on Innovation & technology in computer science education* (pp. 273-278). ACM. https://dl.acm.org/citation.cfm?id=2591748 (1.03.2018)

Downey, A. (2012). *Think Python*. " O'Reilly Media, Inc.". http://lib.hpu.edu.vn/bitstream/handle/123456789/21472/69_thinkpython.pdf?sequence=1 (10.02.2018)

Hansalu, A. (2015). *Programmeerimisülesande lahendamise dünaamika analüüs programmeerimiskeskkonna Thonny logifailide põhjal*. (Magistriõppe lõputöö). Tartu Ülikool, arvutiteaduse instituut.

Helminen, J., Ihantola, P. & Karavirta, V. (2013). Recording and analyzing in-browser programming sessions. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research* (pp. 13-22). ACM. https://dl.acm.org/citation.cfm?id=2526970 (15.03.2018)

Hosseini, R., Vihavainen, A., & Brusilovsky, P. (2014). *Exploring problem solving paths in a Java programming course*. http://d-scholarship.pitt.edu/21832/1/Paper_Pages_from_PPIGproceedings.pdf (23.04.2018)

Introducing JSON. (s.a.). http://www.json.org/ (22.03.2018)

Jadud, M. C. (2005). A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, *15*(1), 25-40. http://jadud.com/dl/pdf/jadud-cse-2005.pdf (11.03.2018)

Jadud, M. C. (2006a). *An exploration of novice compilation behaviour in BlueJ* (Doctoral dissertation, University of Kent). https://core.ac.uk/download/pdf/64181.pdf (11.03.2018)

Jadud, M. C. (2006b). Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research* (pp. 73-84). ACM. https://dl.acm.org/citation.cfm?id=1151600 (11.03.2018)

Jansen, D. & Schuwer, R. (2015). Institutional MOOC strategies in Europe. *Status Report Based on a Mapping Survey Conducted in October-December 2014. Mimeo*. https://www.surfspace.nl/media/bijlagen/artikel-1763-22974efd1d43f52aa98e0ba04f14c9f3.pdf (10.02.2018)

Jordan, K. (2014). MOOC completion rates: the data (2013). *Reference Source*. http://www.katyjordan.com/MOOCproject.html (10.02.2018)

Kiesmüller, U., Sossalla, S., Brinda, T. & Riedhammer, K. (2010). Online identification of learner problem solving strategies using pattern recognition methods. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education* (pp. 274-278). ACM. https://dl.acm.org/citation.cfm?id=1822167 (11.03.2018)

Kodasmaa, R. (2017a). *Programmeerimiskeele Python veateated programmeerimise algõppes* (Magistritöö). Tartu Ülikool, arvutiteaduse inistituut. https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=56952&year=2017 (02.05.2018)

Kodasmaa, R. (2017b). *Levinumad Pythoni veateated programmeerimise algõppes*. https://courses.cs.ut.ee/2017/eprogalused/spring/Main/Veateated (22.03.2018)

Lepp, M., Luik, P., Palts, T., Papli, K., Suviste, R., Säde, M. & Tõnisson, E. (2017). MOOC in Programming: A Success Story. In *ICEL 2017-Proceedings of the 12th International Conference on e-Learning* (p. 138). Academic Conferences and publishing limited. (15.02.2018)

Mackness, J., Mak, S. & Williams, R. (2010). The ideals and reality of participating in a MOOC. In *Proceedings of the 7th International Conference on Networked Learning 2010*. University of Lancaster.

https://researchportal.port.ac.uk/portal/files/108952/The_Ideals_and_Realilty_of_Participating_in_a_MOOC.pdf (17.02.2018)

Magee, J. (2014). *Python Debugging (fixing problems). A guide for beginner programmers*. Bostoni Ülikoolis kasutusel olev programmeerimiskeele Python veateadete materjal. http://www.cs.bu.edu/courses/cs108/guides/debug.html (20.02.2018)

Marceau, G., Fisler, K., & Krishnamurthi, S. (2011a). Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 499-504). ACM. https://dl.acm.org/citation.cfm?id=1953308 (19.05.2018)

Marceau, G., Fisler, K., & Krishnamurthi, S. (2011b). Mind your language: on novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software* (pp. 3-18). ACM. https://dl.acm.org/citation.cfm?id=2048241 (19.05.2018)

Mather, R. (2015). Multivariate Gradient Analysis for Evaluating and Visualizing a Learning System Platform for Computer Programming. *IAFOR Journal of Education*, *3*(1), 17-30. https://eric.ed.gov/?id=EJ1100567 (10.05.2018)

MOOC – miks, kellele ja kuidas? (2017). https://sisu.ut.ee/moocs (20.02.2018)

MOOC Programmeerimise alused. (2017a). Thonny. https://courses.cs.ut.ee/2017/eprogalused/Main/Thonny (15.03.2018)

MOOC Programmeerimise alused. (2017b). 8.2 Arvestusülesanne. https://courses.cs.ut.ee/2017/eprogalused/spring/Main/Arvestusylesanne (9.04.2018)

Onah, D. F., Sinclair, J. & Boyatt, R. (2014). Dropout rates of massive open online courses: behavioural patterns. *EDULEARN14 Proceedings*, 5825-5834. http://wrap.warwick.ac.uk/65543/1/WRAP_9770711-cs-070115-edulearn2014.pdf (20.02.2018)

Pedel, K. (2016). *E-kursuse Programmeerimise alused logifailide analüüs* (Bakalaureusetöö). Tartu Ülikool, arvutiteaduse instituut. https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=53665&year=2016 (23.03.2018)

Pettit, R. S., Homer, J., & Gee, R. (2017). Do Enhanced Compiler Error Messages Help Students?: Results Inconclusive. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 465-470). ACM. https://dl.acm.org/citation.cfm?id=3017768 (21.03.2018)

Pritchard, D. (2015). Frequency distribution of error messages. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools* (pp. 1-8). ACM. https://arxiv.org/pdf/1509.07238.pdf (19.05.2018)

Programmeerimise alused. (2016). https://courses.cs.ut.ee/2015/eprogalused/fall (15.03.2018)

Programmeerimisest maalähedaselt. (2017). https://courses.cs.ut.ee/2017/progmaa/fall (15.03.2018)

Python 3.6.4 documentation. (s.a.). The Python Tutorial: 8. Errors and Exceptions. https://docs.python.org/3/tutorial/errors.html (20.03.2018)

Python Errors. (2012). California Ülikoolis kasutusel olev programmeerimiskeele Python veateadete materjal. https://discover.cs.ucsb.edu/commonerrors/pythonerrors.html (20.02.2018)

Remm, K., Remm, J. & Kaasik, A. (2012). *Ruumiliste loodusandmete statistiline analüüs*. Tartu Ülikooli Ökoloogia ja Maateaduste Instituut. http://dspace.ut.ee/handle/10062/26456 (9.05.2018)

Schliep, P. A. (2015). Usability of Error Messages for Introductory Students. *Scholarly Horizons: University of Minnesota, Morris Undergraduate Journal, 2*(2), 5. http://digitalcommons.morris.umn.edu/cgi/viewcontent.cgi?article=1029&context=horizons (10.03.2018)

Spacco, J., Fossati, D., Stamper, J. & Rivers, K. (2013). Towards improving programming habits to create better computer science course outcomes. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education* (pp. 243-248). ACM. https://dl.acm.org/citation.cfm?id=2465594 (15.03.2018)

Spacco, J., Strecker, J., Hovemeyer, D. & Pugh, W. (2005). Software repository mining with Marmoset: An automated programming project snapshot and testing system. In *ACM SIGSOFT Software Engineering Notes* (Vol. 30, No. 4, pp. 1-5). ACM. https://dl.acm.org/citation.cfm?id=1083149 (15.03.2018)

Vihavainen, A., Luukkainen, M. & Ihantola, P. (2014). Analysis of source code snapshot granularity levels. In *Proceedings of the 15th Annual Conference on Information technology education* (pp. 21-26). ACM. http://dl.acm.org/citation.cfm?id=2656473 (16.03.2018)

Vihavainen, A., Helminen, J. & Ihantola, P. (2014). How novices tackle their first lines of code in an ide: Analysis of programming session traces. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research* (pp. 109-116). ACM. https://dl.acm.org/citation.cfm?id=2674692 (16.03.2018)

Watson, C., Li, F. W. & Godwin, J. L. (2013). Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Advanced learning Technologies (ICALT), 2013 IEEE 13th international conference on* (pp. 319-323). IEEE. http://dro.dur.ac.uk/19225/1/19225.pdf (10.03.2018)

Lisad

I. Graafikuid koostav programm

Programm graafikuteloomine.py asub lisade zip-failis programmid-ja-logifail.zip.

II. Valitud andmeid logist tabelisse kirjutav programm

Programm logisttabelisse.py asub lisade zip-failis programmid-ja-logifail.zip.

III. Andmete tabel

Tabelisse on koondatud kõikide lahenduste detailsed andmed (asub Google Drive'is).

IV. Andmete koondtabel

Tabelisse on koondatud kõikide lahenduste summeeritud andmed (asub Google Drive'is).

V. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, **Heidi Meier**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose
   Õppijate käitumismustrid programmeerimisülesande lahendamisel: logifailide analüüs,

   mille juhendaja on PhD Eno Tõnisson,

   1.1. reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.
3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, **21.05.2018**