TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Merili Pihlak

Koodianalüsaatori arendus kursuse „Programmeerimise alused“ projektide tehnilise analüüsi automatiseerimiseks

Bakalaureusetöö (9 EAP)

Juhendaja:
Reelika Suviste, PhD

Tartu 2025

Koodianalüsaatori arendus kursuse „Programmeerimise alused“ projektide tehnilise analüüsi automatiseerimiseks

Lühikokkuvõte:

Programmeerimise kursusel on ajamahukas analüüsida ja hinnata üle saja projekti. Selle lõputöö käigus arendati koodianalüsaator kursusele „Programmeerimise alused“, et projekte automaatselt analüüsida. Tulemusena valmis koodianalüsaator, mis väljastas kahe minutiga üle saja projekti koodianalüüsid ühte kokkuvõtvasse Exceli faili, kus on igal projektil 48-50 erinevat tehnilist analüüsimõõtu.

Võtmesõnad: Koodianalüsaator, koodianalüüs, programmeerimine, Python

CERCS: P175 Informaatika, süsteemiteooria

Development of a Code Analyzer for Automating the Technical Analysis of the “Introduction to Programming” Course Projects

Abstract:

Evaluating and performing analysis to over 100 projects in a programming course is a timeconsuming task. With this thesis, a code analyzer was proposed and developed for the course “Introduction to Programming” to automatically analyze projects. The result was a code analyzer, which in 2 minutes, outputted code analyses of over a hundred projects into a single summarized Excel file, where each project has 48-50 different technical analysis metrics.

Keywords: Code analyzer, code analysis, programming, Python

CERCS: P175 Informatics, systems theory

Sisukord

1. Sissejuhatus

Staatiline analüüs on „süsteemi või komponendi hindamise protsess selle vormi, struktuuri, sisu või dokumentatsiooni alusel“ [1]. Selle protsessi rakendamiseks koodile kasutatakse tehnilist tööriista, mida nimetatakse staatiliseks koodianalüsaatoriks (edaspidi tähistab koodianalüsaator staatilist koodianalüsaatorit). Analüsaatori eesmärk on märgata reeglite ja tavade rikkumisi võimalikult efektiivselt, tähtsad on seejuures kiirus ja täpsus.

Programmikoodide automaatne analüüs on väga aktuaalne, kuna tarkvara osakaal igapäevaelus on kasvava trendiga ning vigade ennetamine selle võrra olulisem [2, 3]. Vea ennetamine on tähenduselt samaväärne kui eelmääratud reeglite ja tavade järgimine, kuna viga on definitsioonilt nõuete täitmata jätmine[1].

Tartu Ülikooli kursus „Programmeerimise alused“ õpetab programmeerist keeles Python[2]. Selle käigus on osalejatel võimalik teostada ka projekt, mis kajastab õpitud teadmisi. Kuigi projektid ei ole mahult suured, tuleb siiski tulenevalt esituste kogusest analüüsida palju koodi (2024. aasta sügissemestril 109 esitust). Selle peale kulub tohutult aega, kuigi oleks võimalik suur osa tööst automatiseerida, kasutades koodianalüsaatorit. Kursuse projektide analüüsimise ning hindamise lihtsustamiseks valmis käesoleva lõputöö käigus koodianalüsaator Tartu Ülikooli kursusele „Programmeerimise alused“.

Käesolev töö koosneb neljast sisulisest peatükist. Peatükis 2 käsitletakse koodianalüsaatoreid, kursust „Programmeerimise alused“, selle projekte ning olemasolevaid lahendusi. Seejärel uuritakse peatükis 3 erinevaid tehnilisi tööriistu koos alternatiividega, millega luua koodianalüsaator kursusele „Programmeerimise alused“. Selle jaoks võetakse vaatluse alla erinevad tööriistad eeltöötluseks, koodianalüüsiks ning järeltöötluseks. Peatükis 4 seatakse nõuded koodianalüsaatorile ning käsitletakse valminud süsteemi implementatsiooni. Antakse ka ülevaade testide ülesehitusest ning tuuakse välja loodud testid, millega süsteemi kontrollida. Viimases sisulises peatükis 5 analüüsitakse testide tulemusi, kui hästi lähevad need nõuetega kokku, süsteemi piiranguid ning võimalikke edasiarendusi.

[1] https://akit.cyber.ee/term/3485
[2] https://www.python.org/

2. Koodianalüsaator

Koodianalüsaatorite eesmärk on võimalikult efektiivselt reeglite ja tavade rikkumisi ennetada. Selle saavutamiseks on mitmeid eri lahendusi ning optimaalse lahenduse jaoks luuakse tihtipeale süsteem vastavalt kontekstile [4–8].

Selles peatükis käsitletakse kõigepealt kursuse „Programmeerimise alused“ ülesehitust ning seejärel selle kursuse projektide ülesehitust ja hindamisnõudeid. Viimaks uuritakse ka olemasolevaid lahendusi.

2.1 Kursus „Programmeerimise alused“

Lõputöö raames loodi koodianalüsaator kursuse „Programmeerimise alused“ projektide analüüsimiseks. Tegu on sissejuhatava programmeerimise kursusega[3], kus õppetöö toimub programmeerimiskeeles Python. Kursuse vältel õpitakse järgnevaid teemasid[4]:

1. programmeerimise põhimõisted, Pythoni süntaks, väljundi saatmine konsooli, konsoolist sisendi võtmine, muutujad, konstandid, andmetüübid, laused (ingl *statements*) ja avaldised (ingl *expressions*), tehtemärgid, standardteegid ja moodulid, erindid;
2. tõeväärtused, võrdlusoperaatorid, loogilised operaatorid, tingimuslaused, `pass`-lause, erindite püüdmine;
3. funktsioonid, parameetrid, argumendid, väärtuse tagastamine, muutuja skoobid, käivitusjärjekord;
4. `while`-tsüklid, `for`-tsüklid, `break` ja `continue` laused;
5. sõned ja nende töötlemine;
6. failid, failihaldus, kataloogid, failiteed, failide avamine ja sulgemine, failide töötlemine;
7. Pythoni järjendid, nende muudetavus, dünaamiline suurus, heterogeensus, viidad (ingl *pointers*), koopia loomine, järjendite töötlemine;
8. kilpkonnagraafika, `EasyGUI`[5].

[3] https://courses.cs.ut.ee/2024/itp/fall/Main/HomePage
[4] https://courses.cs.ut.ee/2024/itp/fall
[5] https://easygui.sourceforge.net/

Nende teemade eesmärk on edasi anda programmeerimise põhitõed ning seejuures piisav pädevus Pythonis, et lahendada varieeruvaid ülesandeid.

2.2 Kursuse „Programmeerimise alused“ projektid

Projekt on kursuse vabatahtlik osa, mille idee on rakendada õpitud teadmisi. Nagu muu õppetöö kursusel, tuleb ka projekt programmeerida Pythonis. See koosneb formuleeringust, programmist ning tutvustavast videost, mis ühiselt moodustavad kuni 14% (14 punkti) lõpphindest[6]. Järgnevalt on välja toodud projekti programmi nõuded hindamisjuhendist[7]:

- vähemalt kaks omaloodud funktsiooni, mida kasutatakse programmis (2 punkti);
- programm võtab failist või kasutajalt sisendi (2 punkti);
- vähemalt kaks tingimuslauset, millest vähemalt ühel peab olema `else`- või `elif`-plokk (2 punkti);
- vähemalt üks tsükkel muul eesmärgil, kui sisendi lugemine (2 punkti);
- vähemalt üks võimalikult lühike erindit püüdev plokk (1 punkt);
- heade koodimistavade järgimine, nagu tähenduslikud muutuja- ning funktsiooninimed, koodimistava DRY (ingliskeelse väljendi *Don’t Repeat Yourself* lühend [9], mis tähendab otsetõlkes ”ära korda ennast“) (1 punkt);
- video- või slaidiesitlus programmist rühmakaaslastele (2 punkti).

Kokku saab programmi eest 10 punkti, video- või slaidiesitlus annab 2 punkti ning projekti formulatsioon samuti 2 punkti.

2.3 Olemasolevad lahendused

Peale paljude tööde läbitöötamist (umbes 20) leiti, et Puulmanni töö „Pythoni teek programmeerimisülesannete automaatseks testimiseks“ [10] sobib võrdluseks kõige paremini. Töö probleem ning eesmärk on sarnased selle lõputööga ja on selge, mis süsteem töö raames realiseeriti.

---

[6] https://courses.cs.ut.ee/2024/itp/fall/Main/Grading

[7] https://courses.cs.ut.ee/2024/itp/fall/Main/ProjectProgram

Puulmanni töös püstitatud probleemiks oli puudulik automaathindamine ning sellest tulenev suur ajakulu. Puulmanni töö sarnaneb käesoleva lõputööga, kuna eesmärk oli vähendada hindamiseks kuluvat aega. Puulmanni lahenduse eesmärk oli tagasisidestada üliõpilasi nende lahenduse korrektsuses. Selle jaoks arendas Puulmann teegi, mis kontrollib programmi väljundeid etteantud sisenditega. See lähenemine sarnaneb komponenditestimisele[8] (ingl *unit testing*), kus iga komponenti testitakse eraldi, kasutades erinevaid sisendeid. Tegu on tugeva meetodiga programmi kvaliteedi hindamiseks, aga see ei sobi erinevate ülesehitusega projektide hindamiseks, kuna oodatavad väljundid varieeruvad samade sisendite puhul.

Erinevalt Puulmanni tööst, kus uuritavad ülesanded olid kindla püstituse ja oodatud väljundiga, on kursusel „Programmeerimise alused“ kõik projektid erinevad. Seetõttu ei saa hinnata, kas projekt töötab ning kas väljund on korrektne ilma arusaamiseta, kuidas programm töötama peaks. Hindamiseks kasutab kursus „Programmeerimise alused“ hindamisskeemi (vt peatükk 2.2), milles uuritakse, kas valminud programm sisaldab piisavalt tsükleid, tingimuslauseid ja muud seesugust. Selline hindamine kontrollib põhiliselt projekti mahtu ning ei suuda hinnata, kas programm töötab korrektselt. Väljundit ei saa kasutada otsese tagasisidena esitajale ning seega on väljundi sihtrühm õppejõud ning tulevased üliõpilased, kelle lõputöö eesmärk on projekte analüüsida (Puulmanni töös oli väljund suunatud esitajale).

Selleks, et oleks võimalik põhjalikku analüüsi teha, peab väljund sisaldama ka muid andmeid peale hinde. Sealt tuleb ka viimane põhiline erinevus Puulmanni tööst: erinevalt Puulmanni teegist, väljastab käesoleva lõputöö raames valminud koodianalüsaator ka palju teisi mõõte programmikoodide kohta. Seejuures teeb koodianalüsaator staatilist analüüsi ja ei käivita koodi; erinevalt Puulmanni tööst, ei pea veenduma, et käivitatud programm oleks turvaline, sest programmi ei käivitata kunagi.

[8] https://akit.cyber.ee/term/13110

3. Kasutatud tehnoloogiad

Lõputöö käigus valminud süsteem on kirjutatud programmeerimiskeeles Python ning koosneb kolmest põhilisest osast:

1. eeltöötlus — leitakse üles iga esituse Pythoni failid;
2. koodianalüüs — sooritatakse failidele erinevaid tehnilisi analüüse;
3. järeltöötlus — kirjutatakse analüüside tulemused Exceli faili.

Python on väga võimekas ning paindlik programmeerimiskeel ning võimaldab hõlpsasti installeerida ning kasutada teiste inimeste loodud teeke. Lõputöö käigus valminud koodianalüsaatori kõigi kolme põhilise osa juures on etapid, mille lahendamine ilma tehnilise tööriistata oleks keeruline. Nendeks etappideks on:

1. kokkupakitud failide lahtipakkimine ning üldine failihaldus eeltöötluses;
2. koodi parsimine ning analüüside loomine koodianalüüsis;
3. tulemuste faili kirjutamine järeltöötluses.

Selles peatükis seletatakse, miks kasutatakse koodianalüsaatori loomiseks Pythonit, milliseid teeke rakendatakse eeltöötluses, koodianalüüsis ja järeltöötluses ning mis on nende teekide eelised ning eesmärgid selles koodianalüsaatoris.

3.1 Programmeerimiskeel Python

Pythoni eeliseid teiste programmeerimiskeelte ees on selle lõputöö käigus valminud koodianalüsaatori juures mitmeid. Umbes 1990. aastal loodi Python eesmärgiga, et olla lihtne ja liidestatav[9] ning see on väga edukalt neid eesmärke täitnud, olles hetkeseisuga aastal 2025 kasutusel enamustes programmeerimiste algkursustel maailmas [11]. Just sellel põhjusel on programmeerimise õpetamiseks Python kasutusel ka kursusel „Programmeerimise alused“. Lisaks kõigele eelmainitule on Pythonis standardteek `ast`[10], mis võimaldab luua abstraktse süntaksipuu (AST on lühend ingliskeelsele tähendusele *abstract syntax tree*) Pythoni koodist ning seda töödelda. Nende eeliste tõttu on see programmeerimiskeel kõige sobilikum „Programmeerimise alused“ kursuse koodianalüsaatori loomiseks, kuna valminud süsteemi on

[9] https://akit.cyber.ee/term/8877-liidestama
[10] https://docs.python.org/3/library/ast.html

kerge muuta, kursuse õpetamiskeel ja koodianalüsaatori keel on ühised ning sellega tuleb kaasa AST töötlemisvahend.

3.2 Teegid — eeltöötlus

Failihalduseks on Pythonis kaks standardteeki `os`[11] ning `shutil`[12]. Teegiga `os` saab teha failihalduse operatsioone individuaalsetel failidel ning teegiga `shutil` failidel ja failikogudel. Nende kahe teegi koostööl on võimalik enamusi selles lõputöös valminud koodianalüsaatori eeltöötluse toimimiseks vajalikke operatsioone teha ning seetõttu on failihalduseks valitud tööriistad `os` ja `shutil`.

Koodianalüsaatori eeltöötluse toimimiseks on vaja ka lahendust kokkupakitud failide lahtipakkimiseks. Sellegi jaoks on Pythonis standardteek `zipfile`[13], kuid see suudab lahti pakkida ainult ZIP-faile ning sellest ei piisa. Lisaks ZIP-failile, mis hoiustab esitusi, võib omakorda esituste sees olla kokkupakitud faile ning need ei pea olema ilmtingimata ZIP-failid. Kokkupakitud failitüüpe on mitmeid ning selleks, et kõiki erinevaid lahti pakkida, eksisteerivad erinevad teegid. Koodianalüsaatori süsteemis kasutatud teek on `patool`[14], mis ühendab mitu lahtipakkimise vahendit üheks. Alternatiiv oleks `extractcode`[15], aga see on vähem populaarne ning erinevalt `patool`-ist ei ole kindlat juhendit, kuidas Pythoni programmis seda jooksutada.

3.3 Teegid — koodianalüüs

Üks staatilise koodianalüüsi vahend on linter[16]. Sellega saab kontrollida, kas järgitakse häid koodimistavasid ning potentsiaalselt ka muid ise defineerituid reegleid. Lintereid on mitmeid erinevaid[17], igaüks oma nõrkuste ja tugevustega.

---

[11] https://docs.python.org/3/library/os.html
[12] https://docs.python.org/3/library/shutil.html#module-shutil
[13] https://docs.python.org/3/library/zipfile.html#module-zipfile
[14] https://pypi.org/project/patool/
[15] https://pypi.org/project/extractcode/
[16] https://owasp.org/www-project-devsecops-guideline/latest/01b-Linting-Code
[17] https://geekflare.com/dev/python-linter-platforms/

Teek `pylint`[18] on aastal 2004[19] loodud linter. Hetkeseisuga on see juba üle 20-aastane avatud lähtekoodiga[20] teek ning seetõttu tõenäoliselt kõige töökindlam ning arenenuma lahendusega Pythoni linter. Lisaks mainitule on `pylint` ka väga kohandatav[21] ning seetõttu kasutatud ka selle lõputöö raames valmiva koodianalüsaatori osana. See teek annab koodile skoori (reaalarv 0.0-10.0) ning selleks, et skoor väljundist eraldada, on mõistlik kasutada regulaaravaldist. Regulaaravaldiste abil saab otsida sõnedest soovitud osasid ning Pythonis on selle jaoks standardteek `re`[22], mida ka selle lõputöö raames valmiva koodianalüsaatori implementatsioonis kasutatakse.

Selleks, et kinnitada kõikide projektikoodi nõuete täitmist, ei piisa ainult linterist. Linteri põhiülesanne on hinnata heade kooditavade järgimist, mis moodustab kõikidest nõuetest ainult ühe (vt peatükk 2.2). Ülejäänud nõuete kontrollimiseks on vaja hinnata koodi ülesehitust. Selle jaoks on Pythonis standardteek `ast`, millega saab koodist luua abstraktse süntaksipuu (ingl *abstract syntax tree*). Abstraktne süntaksipuu on puu (arvutiteaduse andmetüüp), mis kannab edasi programmi struktuuri, sisu ning käitusjärjekorda. Selle abil on võimalik hinnata, mida programm teeb ning selle töö kontekstis tähtsamalt, millest programm koosneb ning kuidas see on ülesehitatud. Abstraktne süntaksipuu võimaldab hindamisskeemi järgi hinde anda ning on seetõttu kasutusel ka lõputöö käigus valmivas koodianalüsaatoris.

Viimaks on koodianalüüsis kasutusel teek `radon`[23], mis on staatilise koodianalüüsi tegemiseks loodud ning annab mitmeid erinevaid mõõte sisendkoodile. Kasutades teeki `radon`, saab vältida rohkemate teekide installeerimist, kuna annab palju eri mõõte. Teegi `radon` väljundist kasutatakse selle lõputöö raames valminud koodianalüsaatoris järgnevaid mõõte:

- McCabe kompleksus ehk tsüklomaatiline kompleksus (ingl *cyclomatic complexity*);
- Halsteadi mõõdud;
- hooldatavuse indeks (ingl *maintainability index*);

[18] https://www.pylint.org/
[19] https://pypi.org/project/pylint/#history
[20] https://github.com/pylint-dev/pylint
[21] https://docs.pylint.org/
[22] https://docs.python.org/3/library/re.html
[23] https://pypi.org/project/radon/

- teised mõõdud, nagu koodiridade arv, loogiliste koodiridade arv, tühjade ridade arv ja nii edasi.

Mõõtude kohta täpsemalt kasutusjuhendis, mis on kaasas lõputöö käigus valminud koodianalüsaatori lähtekoodiga (vt peatükk 6).

Lisaks eelmainitud teekidele on kasutusel ka teegid `transformers`[24] ning `torch`[25] selleks, et tehisintellekti genereeritud koodi tuvastada. Kasutusel on tehisintellekti mudel, mille eesmärk on koodi genereerida. Kui sellele ette anda valmis kood, siis saab välja võtta selle mudeli üllatuvuse (ingl *perplexity*) mõõdu. Mida rohkem üllatunud on tehisintellekti mudel, seda rohkem tõenäoliselt erineb etteantud kood selle mudeli treeningandmetest ehk millestki, mida see mudel genereeriks. See on aga testimata ning implementeeritud vaid edasiarenduse idee näitlikustamiseks.

3.4 Teegid — järeltöötlus

Viimaks on järeltöötlus, milles on kasutusel teek `pandas`[26]. See on Pythoni teek, mis loodi aastal 2008[27] andmeanalüüsi läbiviimiseks Pythonis. See on väga optimiseeritud ning võimaldab teha keerulisi andmetöötluse operatsioone kiirelt. Teegi `pandas` poolt loodud andmestruktuuri nimetatakse `DataFrame`'iks. See sarnaneb arvutustabelile ning seetõttu pakub `pandas` võimalust kirjutada `DataFrame`'is olevad tulemused otse arvutustabelisse. Teine valik oleks kasutada teeki `XlsxWriter`[28], aga see töötab ainult Exceli failidel (teegiga `pandas` saab muudesse arvutustabeli failitüüpidesse ka kirjutada) ning nõuab iga välja kirjutamist individuaalselt. Teek `pandas` kasutab tegelikult teeki `XlsxWriter`, et kirjutada Exceli faili ning seetõttu pole eelist kasutada lihtsalt `XlsxWriterit`. Teek `pandas` on kasutusel, sest võimaldab teha andmed kergelt `DataFrame`'iks ning seejärel vormistusega kirjutada Exceli faili[29].

[24] https://pypi.org/project/transformers/
[25] https://pypi.org/project/torch/
[26] https://pandas.pydata.org/
[27] https://pandas.pydata.org/about/
[28] https://pypi.org/project/XlsxWriter/
[29] https://stackoverflow.com/a/45832119

4. Metoodika

Selle peatüki eesmärk on välja tuua lõputöö käigus valminud koodianalüsaatori nõuded, arhitektuur ning testimise meetodika. Koodianalüsaatori kasutamise põhjuseks on lihtsustada tehnilise analüüsi protsessi ning seetõttu peab see täitma etteantud funktsionaalseid ja mittefunktsionaalseid nõudeid. Tuues välja süsteemi arhitektuuri, on võimalik näha, kuidas nõuete täitmist teostatakse ning läbimõeldud testimisega saab tõestada nõuete täitmist.

4.1 Nõuded koodianalüsaatorile

Funktsionaalsete nõuete eesmärk on seada ootused, kuidas süsteem toimima peaks. Mittefunktsionaalsed nõuded on selleks, et määratleda süsteemi võimekused ning piirangud.

Selle töö käigus valminud koodianalüsaatori funktsionaalsed nõuded on:

- lihtne seadistamine — seadistamine peab peale allalaadimise sisaldama ülimalt viite sammu;
- mugav kasutamine 1 — analüüs peab olema sooritatav ühe käsuga, mis võtab valikulisteks argumentideks sisend- ning väljundfailiteed;
- mugav kasutamine 2 — kui failiteed pole antud, siis kasutab vaikeväärtuseid `submissions.zip` ning `output.xlsx` vastavalt sisend- ning väljundfailiteedeks;
- mugav kasutamine 3 — koodianalüsaator peab automaatselt tuvastama iga projekti Pythoni faili ning neile analüüsi sooritama. Kõik projektid on ühes ZIP-failis, igaühel oma kaust, mis sisaldab esituse faile;
- kasulik väljund 1 — väljund peab sisaldama hinnet projektidele (vastavalt hindamisskeemile) ning vähemalt 25 erinevat mõõtu, mille põhjal saaks edasisi analüüse teha;
- kasulik väljund 2 — väljund peab olema kirjutatud arvutustabelisse (ingl *spreadsheet*), eelistatavalt Exceli formaadis, nii et oleks selgesti aru saada, millisele projektile, millised tulemused vastavad;
- valikuliselt võib linkida arvutustabeli projektide tulba projektid nendele vastavate Pythoni failidega.

Mittefunktsionaalsed nõuded on:

- kiirus — analüüs sajale projektile peab kestma ülimalt 5 minutit;

- töökindlus — analüüs ei tohi tööd lõpetada ühe puuduliku või vigase projektikoodi pärast;
- täpsus — analüüsi tulemused peavad olema korrektsed ning täpsed kuni veaga ±1%.

Nende nõuete täitmiseks loodi selle töö raames koodianalüsaator, mille arhitektuur on käsitletud järgmises alampeatükis 4.2.

4.2 Koodianalüsaatori arhitektuur

Peatükis 3 mainiti, et koodianalüsaator koosneb kolmest põhilisest osast: eeltöötlus, koodianalüüs ning järeltöötlus. Vaatluse all olev koodianalüsaator on deterministlik ning ülesehitatud modulaarselt. Viimane tähendab, et iga põhilise osa jaoks on loodud eraldi moodul. Moodul on funktsioonide ja muude seesuguste asjade kogum, mida saab kasutada kui tehnoloogilisi tööriistu [12]. Moodul erineb teegist selle poolest, et teek on moodulite kogum.

4.2.1 Eeltöötlus

Eeltöötluse moodul pakub kahte põhilist funktsiooni. Nendeks on `extract_recursive` ning `find_python_file`. Esimene on kokkupakitud arhiivi rekursiivseks lahtipakkimiseks ning teine on kaustast rekursiivselt Pythoni failide otsimiseks. Rekursiivne lahtipakkimine toimib järgnevalt:

1. kui tegu on kaustaga, siis rakendada `extract_recursive` funktsiooni kõikidele selle kausta sees olevatele failidele;
2. vastasel juhul kontrollida, kas failitüüp on kokkupakitud failitüüp;
3. kui ei ole, siis lõpetatakse töö;
4. kui tegu on kokkupakitud failiga, siis luuakse nimi sihtkaustale. Nimeks on kokkupakitud faili nimi ilma faililaiendita (ehk arhiivi `submissions.zip` sihtkaust on `submissions`);
5. kontrollitakse, kas sihtkaust on juba olemas;
6. kui ei ole, siis pakitakse lahti otse sihtkausta;
7. kui sihtkaust on juba olemas, siis tehakse ajutine kaust, millesse pakitakse sisu lahti (ajutise kausta nimi on sihtkausta nimi + katse number, kus katse number on mitmendat korda üritatakse luua unikaalne ajutise kausta nimi);
8. lahti pakitud kaustas rakendatakse `extract_recursive` funktsiooni kõigile failidele;

9. kui sihtkaust oli eelnevalt juba olemas, siis see kustutatakse ning ajutine kaust nimetatakse ümber sihtkaustaks;
10. juhul kui eelmine samm ebaõnnestus, kuna ei saanud ajutist kausta ümber nimetada sihtkaustaks, siis kasutatakse edaspidi ajutist kausta sihtkaustana;
11. töö lõpetatakse ning tagastatakse sihtkausta nimi.

Kui Moodle'i süsteemist alla laadida üliõpilaste esitused, siis tulevad need ühes ZIP-failis nagu joonisel 1. Joonisel tähistavad `XXXXXXX` ja `YYYYYYY` 7-kohalisi täisarve; `N` ja `M` on täisarvud, mis tähistavad järjekorranumbrit; failis `onlinetext.html` on esitajate lisatud tekst/kommentaar HTML-faili kujul. Kaustas lõpuga `_assignsubmission_file` on üliõpilase üleslaetud failid.

```
submissions.zip
    ├ eesnimi1_perenimi1_XXXXXXX_assignsubmission_file
    |   ├ esituse fail/kaust 1
    |   ├ ...
    |   └ esituse fail/kaust N
    ├ eesnimi1_perenimi1_XXXXXXX_assignsubmission_onlinetext
    |   └ onlinetext.html
    |
    ├ ...
    |
    ├ eesnimiM_perenimiM_YYYYYYY_assignsubmission_file
    |   └ ...
    └ eesnimiM_perenimiM_YYYYYYY_assignsubmission_onlinetext
        └ ...
```

Joonis 1. Üliõpilaste esituste failistruktuur kokkupakitud arhiivis.

Selleks, et teha koodianalüüsi projektidele, tuleb leida projektide koodid. Kaustades lõpuga `_assignsubmssion_onlinetext` pole projektifaile ning neid peab ignoreerima. Seega tuleb üles leida projektide kaustad ehk need, mille lõpp on `_assignsubmission_file`. Seejärel tuleb iga projekt panna paari tema sees olevate Pythoni failidega. Üles tuleb otsida kõik, kuna on juhtumeid, kus uuendatud kood esitatakse koos esialgse koodiga. Vahepeal laetakse üles faile, mille faililaiend ei ole korrektne (näiteks Pythoni koodi esitamiseks TXT-fail või täiesti ilma faililaiendita fail). Need on küll erandid aga neid ei tohi ignoreerida. Pythoni failide

leidmine koosneb kahest põhilisest funktsioonist ning kolmandast abifunktsioonist. Esimene funktsioon töötab järgneva loogikaga:

1. otsi üles kõik failid, mis lõppevad laiendiga `.py` ning mille sisu on Pythoni kood;
2. kui ei leidnud midagi, siis otsi üles kõik failid, mis sisaldavad sõne `py` ning mille sisu on Pythoni kood;
3. kui ei leidnud midagi, siis otsi üles kõik failid, mille sisu on Pythoni kood;
4. tagasta tulemused (kui midagi ei leitud, siis tagastatakse lihtsalt tühi järjend).

Otsimine toimub rekursiivselt ning selle jaoks kasutatakse teist funktsiooni. Selle loogika on järgnev:

1. kui tegu on failiga ning fail vastab tingimustele (näiteks, kas sisaldab sõne `py` ning on Pythoni kood), siis lisada tulemuste hulka;
2. kui tegu ei ole failiga (ehk tegu on kaustaga), siis rakendada teist funktsiooni rekursiivselt kõigile selle sees olevatele failidele ning saadud tulemused kokku koguda;
3. tagastada tulemused.

Lõpuks on kolmas abifunktsioon, mille eesmärk on kontrollida, kas faili sisu on Pythoni kood. Selle jaoks avatakse fail ning üritatakse teha sellest Pythoni abstraktne süntaksipuu. Kui tegu on Pythoni failiga, siis see operatsioon on edukas ning erindit ei viska. Seetõttu tagastatakse tõene tõeväärtus, mis tähistab, et tegu on Pythoni failiga. Kui tegu ei ole Pythoni failiga, siis see operatsioon viskab erindi ning selle kinnipüüdmisel saab tagastada väära tõeväärtuse, teatamaks, et tegu ei ole Pythoni failiga.

Nende kolme funktsiooniga on võimalik kõikides projektides programmikoodid üles leida. Seejärel tuleb sooritada koodidele analüüsid, mille jaoks luuakse koodianalüsaatori objekt. Käiakse läbi kõik leitud Pythoni failid, igaühele sooritatakse koodianalüsaatori objektiga analüüs ning tulemused kogutakse järjendisse.

4.2.2 Koodianalüüs

Koodianalüüs on selle lõputöö raames valminud süsteemi kõige mahukam osa ning koosneb mitmest osast. Järgnev on koodianalüüsi pinnapealne selgitus:

1. loetakse koodifail sõnena sisse ning luuakse abstraktne süntaksipuu;
2. arvutatakse kogu koodi terminite suhtes terminite osakaal, mida kursuse vältel ei õpitud;

3. arvutatakse hindamisskeemi järgi hinne, muud mõõdud ning teegi `pylint` skoor;
4. tulemused pannakse koos failiteega sõnastikku (vt joonis 2), kus `file_path` on Pythoni faili asukoht, `pylint_score` on teegi `pylint` antud skoor, `unfamiliar_terms` on mitteõpitud terminite osakaal, `points` on hindamisskeemi järgi antud hinne ning `details` on sõnastik, mis sisaldab ülejäänud analüüsitulemusi.

```
final_details = {
    "file_path": file_path,
    "pylint": pylint_score,
    "unfamiliar_terms": unfamiliar_terms,
    "points": points,
    "details": details,
}
```

Joonis 2. Ühe faili koodianalüüsi tulemuste sõnastiku ülesehitus.

Mitteõpitud terminite osakaal arvutatakse kasutades kahte hulka. Esimene hulk on õpitud terminid kursusel (manuaalselt õppematerjalidest kogutud, vt joonis 3). Teine hulk on programmikoodis loodud funktsioonid ja muutujad, mis leitakse koodi abstraktse süntaksipuu läbimisega. Mõlema hulga ühend omistatakse uude muutujasse. Käiakse koodi abstraktne süntaksipuu korra veel läbi ning loetakse kokku terminite koguarv ning tundmatute terminite koguarv (terminid, mis ei kuulu õpitud ega programmikoodis loodud terminite hulka). Lõpuks jagatakse tundmatute arv terminite koguarvuga ning sedaviisi saadakse osakaal.

Teegiga `ast` läbitakse eelnevalt loodud abstraktne süntaksipuu, mille käigus loetakse kokku erinevate terminite esinemissagedused koodis. Terminite kokkulugemine on üldiselt sama protsess iga terminiga: tuleb kontrollida, mis tüüpi terminiga on tegu ning seejärel saab võtta vastavast väljast selle nime. Põhiline keerukus seejuures on tsüklite ning loodud funktsioonide kokkulugemine, kuna mõlemale on hindamisskeemis (vt peatükk 2.2) seatud eritingimused. Tsüklite arvestamiseks on oluline, et need täidataksid lisaks faili või sisendi lugemisele ka muud ülesannet ja funktsioonide juures peab jälgima, et need oleksid kasutusel.

Selleks, et hinnata, kas tsükkel teeb midagi muud peale failist lugemise, on kasutusel abimeetodid. Esimene abimeetod tagastab, kas termin on konsoolist lugemine, faili avamine, faili lugemine või faili sulgemine. Teine abimeetod kontrollib rekursiivselt, kas kõik terminid ning nendesisesed terminid on ainult failist või konsoolist lugemiseks (terminite nagu `if`-lause puhul ignoreeritakse

```
known_terms = {
    "print","for","in","range","if","else","elif","int","str","bool",
    "float","input","True","False","None","list","set","tuple","complex",
    "dict","type","while","break","continue","return","pass","import",
    "from","not","or","and","is","math","pow","try","except","quit",
    "ValueError","def","random","randint","sqrt","factorial","log",
    "global","NameError","SyntaxError","isupper","islower","IndexError",
    "TypeError","upper","lower","title","capitalize","len","split",
    "join","lstrip","rstrip","isnumeric","isdigit","enumerate","open",
    "close","read","readline","readlines","write","writelines",
    "FileNotFoundError","append","pop","index","extend","insert",
    "remove","sort","reverse","max","min","sum","copy","turtle",
    "forward","left","exitonclick","setup","bgcolor","title","color",
    "pensize","right","penup","pendown","shape","speed","stamp",
    "circle","end_fill","easygui","msgbox","buttonbox","choicebox",
    "enterbox","integerbox"
}
```

Joonis 3. Kursusel õpitud terminite hulk

neid ennast ning kontrollitakse ainult nende keha). Teist abimeetodit rakendatakse kontrollitava tsükli kehale. Kui ükski tulemustest on väär, siis leidub mingi muu termin peale sisendi lugemise. Sellisel juhul on tegemist tsükliga, millel on muu eesmärk kui sisendit lugeda (algoritm on joonisel 4).

Funktsioonide kontrolliks peab koguma kokku kõik funktsiooni definitsioonid (koht koodis, kus funktsioon defineeriti) ning funktsiooni väljakutsumised (funktsioonide kasutamine). Need kaks tuleb koguda eri hulkadesse. Jälgides, millised funktsiooni definitsioonid ei ole funktsiooni väljakutsumistes, saab arvutada väljakutsumata funktsioonide arvu (vt joonis 5).

Selleks, et hindamisskeemi järgi projekte hinnata, on vaja mitmed terminid kokku lugeda. Ülejäänud terminite kokkulugemine on intuitiivsem (tuleb lihtsalt kokku lugeda esinemiste arv) ning seega sellel teemal rohkem ei peatu. Lisaks hindele antakse terminite arvud mõõtudena kaasa (täpsemalt kasutusjuhendis).

Järgmine samm on `pylinti` analüüs. Teek `pylint` sooritab analüüsi seadistuse järgi. Kursusel „Programmeerimise alused“ on enamus osalejatest algajad ning seetõttu ei saa eeldada neilt kõikide heade koodimistavade järgimist. Kursuse eesmärk on alused selgeks teha ning põhjalik

```
def fun(loop_body: list[ast.stmt]) -> bool:
    def abimeetod1(node: ast.AST) -> bool:
        return (is_input_node(node) or
                is_file_opening_node(node) or
                is_file_reading_node(node) or
                is_file_closing_node(node))

    def abimeetod2(stmt: ast.stmt) -> bool:
        if isinstance(stmt, (ast.Break, ast.Continue, ast.Pass,
                             ast.Import, ast.ImportFrom)):
            return True
        # Assign: a = fun()
        # Expr: fun()
        if (isinstance(stmt, ast.Assign) or
            isinstance(stmt, ast.Expr)):
            return (isinstance(stmt.value, ast.Call) and
                    abimeetod1(stmt.value))

        if isinstance(stmt, ast.With):
            return (all(abimeetod1(item.context_expr)
                        for item in stmt.items) and
                    all(abimeetod2(s) for s in stmt.body))

        if isinstance(stmt, ast.If):
            return (all(abimeetod2(s) for s in stmt.body) and
                    all(abimeetod2(s) for s in stmt.orelse))

        if isinstance(stmt, (ast.For, ast.While, ast.Try)):
            return all(abimeetod2(s) for s in stmt.body)

        return False

    return not all(abimeetod2(stmt) for stmt in loop_body)
```

Joonis 4. Funktsioon, et leida, kas tsükkel on ainult faili või sisendi lugemiseks.

arusaam headest tavadest tuleb peamiselt kogemusest. Seetõttu on välja lülitatud 37 reeglit, mida `pylint` jälgib. Ülejäänutega arvutab `pylint` skoori (reaalarv 0.0-10.0), mida kasutatakse hindamisskeemis koodimistavade punkti andmiseks. Kui teegi `pylint` skoor ei ole väiksem

```
unique_calls = set(func_calls)

total_defs_not_called = 0

for _def in func_defs:
    if _def not in unique_calls:
        total_defs_not_called += 1
```

Joonis 5. Väljakutsumata funktsioonide arvu leidmine.

kui 7, siis antakse punkt. Lisaks loetakse kokku 15 reegli rikkumiste arvud `pylinti` väljundis ning lisatakse `details` sõnastikku. Põhjalikumalt nende kohta kasutusjuhendis.

Viimaseks analüüsi osaks on teegi `radon` väljundite kogumine ning detailide hulka lisamine. Teegi `radon` neli põhilist väljundit on hooldatavuse indeks, tsüklomaatiline kompleksus, Halsteadi analüüs ning tooranalüüs (ingl *raw analysis*). Igale väljundile on eraldi analüüsivahend. Hooldatavuse indeksi ülesanne on näidata, kui hooldatav on lähtekood (projektide Pythoni failid selle lõputöö raames). Hetkeseisuga on see aga väga eksperimentaalne mõõt ning seda ei soovitata jälgida sama tähtsusega kui teisi teegi `radon` mõõte[30]. Sellegipoolest on see implementeeritud koodianalüsaatori väljundisse, kuna võib anda hilisemas analüüsis kasulikku informatsiooni projektide kohta.

Järgmine mõõt, mille `radon` annab, on tsüklomaatiline keerukus. See on võrdne koodiplokis olevate hargnemiskohtade arvuga pluss üks. Harngemiskohtadeks on `if`-, `elif`-, `for`-, `while`-, `except`-, `with`- ja `assert`-laused (`else`-lause ei ole, kuna ei loo uut hargnemist), tõeväärtusoperaatorid `and` ja `or` ning järjendi arusaamise avaldis (ingl *list comprehension*, vt joonis 6). Teek `radon` annab tsüklomaatilise keerukuse igale koodiplokile eraldi. Projektid on aga erinevate pikkustega ning plokkide arvudega, seega ei saa neid otse võrrelda. Lahendamaks seda mure arvutatakse selle lõputöö raames valminud koodianalüsaatoris iga projekti puhul nende tsüklomaatilise keerukuse kogusumma, keskmine ning standardhälve (vt jooniseid 7 ja 8).

[30] https://radon.readthedocs.io/en/latest/intro.html#maintainability-index

```
X = [1, 5, 2, 3]
# A
A = [elem - 1 for elem in X]
# B
B = X.copy()
for i, elem in enumerate(X):
    B[i] = elem - 1
```

Joonis 6. Näide lihtsast järjendi arusaamise avaldisest (A) koos peaaegu samaväärse tsükliga selle järel (B).

$$S = \sum_{i=1}^{n} f(i)$$

$$m = \frac{S}{n}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(f(i) - m)^2}{n}}$$

Joonis 7. Tsüklomaatilise keerukuse kogusumma S, keskmise m ning standardhälbe σ väärtuste arvutamine. n on plokkide arv, $f(i)$ on ploki i tsüklomaatiline keerukus.

```
cc = [res.complexity for res in cyclomatic_analysis_results]
cc_total = sum(cc) # kogu keerukus
cc_mean = cc_total / max(1, len(cc)) # keskmine keerukus
cc_deviations_squared = [(complexity - cc_mean)**2 for complexity in cc]
cc_variance = sum(cc_deviations_squared) / max(1, len(cc))
cc_std = sqrt(cc_variance) # standardhälve
```

Joonis 8. Kood tsüklomaatilise keerukuse kogusumma, keskmise ning standardhälbe väärtuste arvutamiseks.

Halsteadi mõõdud on kõik arvutatud nelja muutuja `n`, `m`, `N` ja `M` põhjal, kus `n` on unikaalsete operaatorite arv, `m` on unikaalsete operandide arv, `N` on operaatorite koguarv, `M` on operandide koguarv. Nende muutujate põhjal on arvutatud järgmised mõõdud[31]:

[31] https://radon.readthedocs.io/en/latest/intro.html#halstead-metrics

- programmi sõnastiku suurus ($n + m$);
- programmi pikkus ($N + M$);
- arvutatud programmi pikkus ($n \log_2 n + m \log_2 m$);
- volüüm ($N \log_2 (n + m)$);
- raskusaste ($\frac{n}{2} * \frac{M}{m}$);
- pingutus (*raskusaste* $*$ *volüüm*);
- eeldatav aeg sekundites, et programmi kirjutada ($\frac{pingutus}{18}$);
- kohaletoimetatud vigade arv ($\frac{volüüm}{3000}$).

Halsteadi mõõtude tähendus ei ole intuitiivne ning nagu teiste mõõtudega, seletatakse need paremini lahti kasutusjuhendis.

Viimaks annab teek `radon` toormõõdud[32], milleks on koodiridade arv, loogiliste ridade arv koodis (iga loogiline rida sisaldab täpselt ühte lauset), lähtekoodi ridade arv, kommentaaridega ridade arv, ainult kommentaaridega ridade arv, mitmerealiste sõnede ridade arv ning tühjade ridade arv. Põhjalikumalt käsitletakse neid kasutusjuhendis.

Lisaks sooritab koodianalüsaator ka tehisintellekti kontrolli, kus kontrollitake `if __name__ == "__main__":` ja `main()` sisalduvust koodis ning leitakse tehisintellekti mudeli üllatuvuse mõõt. See on aga eksperimentaalne lahendus ning võimalus edasiarenduseks.

4.2.3 Järeltöötlus

Järeltöötluse esimene osa on saadud analüüside tulemused teha üheks sõnastikuks, kus võtmeteks on mõõtude nimetused ning väärtusteks on mõõtude väärtused (välja arvatud `file_path` tulp, kus on väärtuseks iga programmikoodi failitee). Kui eelnevalt oli tulemuste sõnastikus omakorda sõnastik `details`, siis selle protsessi käigus viiakse kõik sõnastiku `details` võtme-väärtuse paarid üle tulemuste sõnastikku ning `details` võtmest saadakse lahti. Seejärel luuakse teegi `pandas` andmeobjekt `DataFrame` tulemustest ning kõik `file_path` tulba failiteed

[32] https://radon.readthedocs.io/en/latest/intro.html#raw-metrics

vormistatakse ümber Exceli hüperlingi (ingl *hyperlink*) vormi (kui esialgne failitee on `X`, siis Exceli hüperlingina on see `=HYPERLINK(X, X')`, kus `X'` on lühendatud failitee).

Järgmine samm järeltöötluses on luua teegi `pandas` objekt `ExcelWriter`, millega saab kirjutada Exceli faile. `DataFrame` kirjutatakse Exceli faili, esimene rida (mõõtude nimed) vormistatakse päiseks. Seejärel vormistatakse juba sisestatud andmed liigendtabeliks[33] (ingl *pivot table*), mis on Excelis arvutustabelisisene tabel, et analüüsida andmeid[34]. Eelviimasena seatakse kõikide tulpade laiused piisavalt suureks, et mahutada päises olevad nimed ära. Lõpuks seatakse esimese ehk failiteede tulba laius suuremaks ning vormistatakse hüperlingi välimuseks. Nende protsesside käigus valmib tulemustest väljundifail.

4.3 Testimine

Testimine on tähtis osa tarkvarast, et kinnitada selle töökindlust ning korrektsust. Selle lõputöö käigus valminud koodianalüsaatori testimine toimib sarnaselt komponenditestimisele. Erinevus tavalisest komponenditestimisest, kus kirjutatakse testijooksutamiskood igale komponendi testijuhule, toimub selle koodianalüsaatori testimine ühe testiga. Selle testi käigus antakse testsisend ning võrreldakse koodianalüsaatori väljundit oodatud väljundiga. Testsisend on ZIP-fail, mis koosneb projektifailidest, aga erinevalt tavaprojektidest on iga projekti eesmärk olla komponenditest. Sedaviisi saab korraga testida terve tarkvara töövoogu ja komponente ükshaaval. Testid igale nõudele on järgmised:

- lihtne seadistamine — manuaalne testimine;
- mugav kasutamine 1 — manuaalne testimine;
- mugav kasutamine 2 — testid kontrollimaks, et vaikeväärtused toimivad;
- mugav kasutamine 3 — selle katavad komponenditestid;
- kasulik väljund 1 — manuaalne testimine;
- kasulik väljund 2 — selle katavad komponenditestid;
- arvutustabeli projektide tulba lingid — manuaalne testimine;

[33] https://akit.cyber.ee/term/2297-pivot-table

[34] https://support.microsoft.com/en-us/office/create-a-pivottable-to-analyze-worksheet-data-a9a84538-bfe9-40a9-a8e9-f99134456576

- kiirus — test kontrollimaks, kas saja või rohkema projekti analüüs kestab ülimalt 5 minutit;
- töökindlus — selle katavad komponenditestid;
- täpsus — selle katavad komponenditestid.

Selleks, et komponenditestid võimalikult palju kataksid võimalusi, peab neid olema mitmeid. Loodud komponenditestid on välja toodud tabelites 1, 2 ja 3.

Lõputöö käigus valminud koodianalüsaator viib läbi staatilist koodianalüüsi, seega koodi ei käivitata. Kõik analüüsitud failid tehakse esimese asjana Pythoni abstraktseteks süntaksipuudeks teegiga `ast`. Kui tegu ei ole Pythoni koodiga, siis tõstab teek `ast` erindi ning faili edasi ei analüüsita. Seetõttu pole vajalik sisendvalideerimist[35] ohtude suhtes eraldi teha.

Tabel 1. Komponenditestid 0-4 lõputöös valminud koodianalüsaatorile

Testi nr	Testi kirjeldus	Oodatud tulemus
0	ZIP-failis mingi muu kaust/fail, mis pole esituse kaust.	Ignoreerida.
1	Tühi projekt.	Ignoreerida.
2	Projekt, mis sisaldab mitut Pythoni faili eri kaustades ning kokkupakitud failides.	Väljundfailis rida igale `.py` lõpuga Pythoni failile.
3	Test nr 2, aga ilma `.py` lõpuga failideta.	Väljundfailis rida igale `py` sisaldavat failile, mis on ka Pythoni sisuga fail.
4	Test nr 3, aga ilma `py` sisaldavate failideta, mis on ka Pythoni sisuga failid.	Väljundfailis rida igale Pythoni sisuga failile.

[35] https://akit.cyber.ee/term/1405-input-validation

Tabel 2. Komponenditestid 5-10 lõputöös valminud koodianalüsaatorile

Testi nr	Testi kirjeldus	Oodatud tulemus
5	`def func1`, `def func2`, 3x `print()`, 2x `func1()`, `func2`	0 punkti
6	`def func1`, `def func2`, 3x `print()`, `func1()`, `func2()`	2 punkti
7a	`f = open()`, `f.read()`, `f.close()`	2 punkti
7b	`with open() as f: f.read()`	2 punkti
7c	`f = open()`, `f.write()`, `f.close()`	0 punkti
7d	`with open() as f: f.write()`	0 punkti
8a	`input()`	2 punkti
8b	`a = input()`	2 punkti
9	2x `if`	0 punkti
10	2x `if`, 1x `elif`	2 punkti

Teegid `os`, `shutil`, `patool`, `re`, `ast`, `pylint`, `radon` ja `pandas` on populaarsed avatud lähtekoodiga lahendused ning põhjalikult testitud[36]. Seetõttu eeldame, et nende väljundid on korrektsed ning ei testi neid eraldi.

[36]Teekide viimatiste (seisuga 14.05.2025) väljalasete testide tulemuste lingid:

- `os`, `shutil`, `re`, `ast`: https://github.com/python/cpython/commit/6280bb547840b609feedb78887c6491af75548e8
- `patool`: https://github.com/wummel/patool/commit/1cdb692b434dd08faeb77326947476abac944f6a
- `pylint`: https://app.codecov.io/gh/pylint-dev/pylint
- `radon`: https://coveralls.io/github/rubik/radon
- `pandas`: https://app.codecov.io/gh/pandas-dev/pandas

Tabel 3. Komponenditestid 11-15 lõputöös valminud koodianalüsaatorile

Testi nr	Testi kirjeldus	Oodatud tulemus
11	2x `if`, 1x `else`	2 punkti
12	Vaata joonist 9.	4 punkti
13a	Sama, mis test 12, aga kommentaar on asendatud vastava lausega.	6 punkti
13b	Sama, mis test 12, aga kommentaar on asendatud vastava lausega.	6 punkti
14	`try except`	1 punkt
15	Teegi `pylint` skoor ei ole väiksem kui 7.	1 punkt

```
while True:
    # Random comment
    # print("") #13a
    for i in range(10):
        # print("") #13b
        with open("a", "r") as f:
            b = f.read()
        if True:
            b = input("")
            break
        else:
            break
    if True:
        b = input("")
        break
    else:
        break
```

Joonis 9. Testid 12, 13a, 13b

5. Tulemused

Põhilise tulemusena valmis lõputöö käigus koodianalüsaator. Selle eesmärk on lihtsustada õppejõudude ning tulevaste lõputöö tegijate tööd, pakkudes vahendit, mis sooritab suure koguse analüüse automaatselt ära, jättes hindajate ülesandeks analüüsida vaid koodianalüsaatori väljundit.

5.1 Testide tulemused

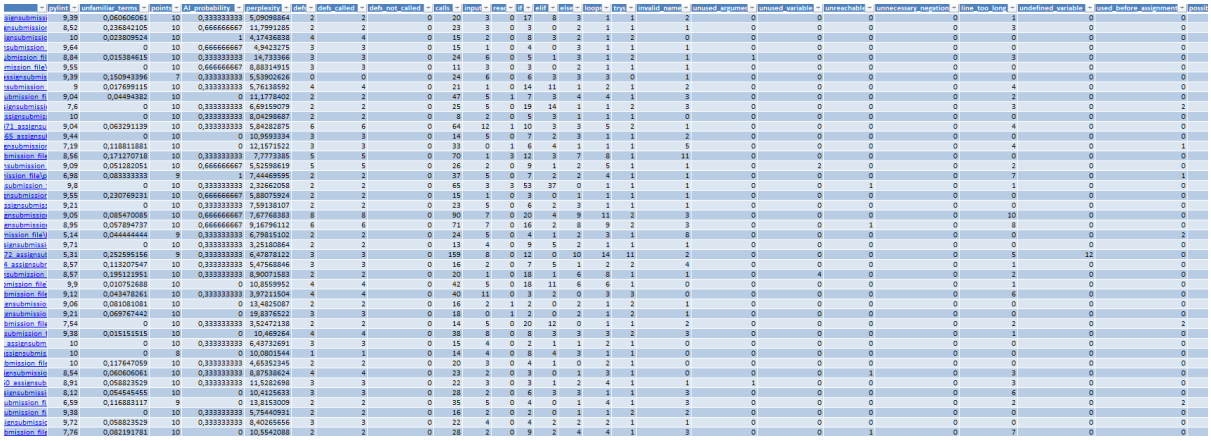
Valminud koodianalüsaator läbis kõik testid edukalt. Järgnevalt on välja toodud funktsionaalsete nõuete täituvused:

- lihtne seadistamine — peale allalaadimise koosneb seadistamine ainult installeerimisskripti käivitamisest eeldusel, et Python on juba süsteemi installeeritud;
- mugav kasutamine 1 ja 2 — analüüsi sooritamine toimub jooksutamisskripti käivitamisega, kuhu saab valikuliselt kaasa anda argumendid `--input` ja `--output` tähistamaks sisend- ja väljundfailiteesid. Kui argumente kaasa ei anta, siis kasutatakse vaikeväärtusi `submissions.zip` ning `output.xlsx`;
- mugav kasutamine 3 — koodianalüsaator leiab iga projekti Pythoni failid rekursiivselt üles, seejuures isegi kui faililaiend pole `.py`;
- kasulik väljund 1 ja link — väljund sisaldab linki projektifailile, hinnet hindamisskeemi järgi ning 50 (48, kui tehisintellekti ei kasutata) erinevat mõõtu, mille põhjal on võimalik edasisi analüüse teha;
- kasulik väljund 2 — väljund on kirjutatud Exceli arvutustabelisse liigendtabelina (vt kujutis 10), et omakorda lihtsustada analüüsimist.

Ning mittefunktsionaalsete nõuete täituvused:

- kiirus — 109 projekti analüüs kestab umbes 2 minutit;
- töökindlus — analüsaator otsib sobilikke faile ning selle käigus väldib mittesobilikke, tagades vigaste failide ignoreerimise;
- täpsus — testitud analüüsi tulemused on 100% täpsed.

Testimiste tulemused viitavad tugevalt edukale lahendusele.



Joonis 10. Näide koodianalüsaatori väljundist. Tulp `file_path` on andmekaitsenõuete tõttu poolik ning näidatud on ainult 43 esitust ja 24 mõõtu.

Lisaks nõuete testimisele võrreldi ka lõputöö raames valminud koodianalüsaatori väljundit hindajate poolt antud hinnetega. Lisaks kümnele punktile, mida oli võimalik programmikoodi eest saada, anti kaks punkti esitluse eest. Kuna esitlust koodianalüsaator hinnata ei saa, siis võrreldi hindaja poolt antud hinnet koodianalüsaatori hindega, ignoreerides esitluse punkte. Tulemusena leiti, et:

- hindajad olid kõikidele projektidele andnud heade tavade järgimise punkti, aga koodianalüsaator seitsmel korral ei andnud (manuaalsel uurimisel selgus, et viiel kehvema `pylinti` skooriga projektil oli punktide eemaldamine õigustatud ning kahel ülejäänul oleks see olnud võimalik, aga mitte ilmtingimata vajalik);
- hindajad olid andnud kolmel korral liigseid punkte täitmata nõuete eest (näiteks tsüklit koodis polnud, aga kaks punkti anti sellegipoolest);
- hindajad olid ühe korra võtnud ühe punkti ära, kuna tsükkel ei teinud midagi muud peale sisendi lugemise (manuaalsel uurimisel selgus, et lisaks sisendi lugemisele tegi tsükkel ka andmetöötlust);
- kümnel korral olid hindajad ära võtnud punkte segase või vigase programmi loogika eest (seda ei saa koodianalüsaator hinnata);
- kahel korral olid hindajad ära võtnud punkte liiga lihtsa programmi eest, kuigi kõik hindamisskeemi nõuded olid täidetud;
- kolmel korral olid hindajad ära võtnud ühe punkti ilma põhjuseta ning selgitusena öelnud, et kõik nõuded on rahuldatud;

- ühel korral polnud hindajad punkte andnud, aga selgitusse olid kirjutanud, et kõik nõuded on rahuldatud;
- kahel korral olid hindajad punkte ära võtnud arvatavasti hilise esituse tõttu, kuna selgituses oli kirjutatud, et kõik nõuded on rahuldatud;
- neljal korral olid hindajad punkte ära võtnud, kuna funktsioonikasutus oli kehv, kuigi nõuded olid täidetud;
- ühel korral olid hindajad kõik punktid ära võtnud tugeva tehisintellekti kahtluse tõttu.

Kokkuvõtvalt andis koodianalüsaator vähemalt 102/109 projekti hinnet õigete punktidega. Hindamisviga hindajate poolt leidus seitsme kuni neljateistkümne projekti seas (vahemik tuleb sellest, et jääb subjektiivseks, mida loetakse hindamisveaks). Kolmteist projekti kaotasid punkte, kuna olid kas segased, vigased, esitatud hilja või tehisintellektiga loodud (koodianalüsaator neid hinnata ei saa).

Saadud tulemused toetavad koodianalüsaatori tähtsust ning vajadust, kuna toovad välja hindamisvead ning segased kohad hindamisskeemis. Näiteks selgus selle analüüsi põhjal vaid hindest (ning hindega kaasnevatest mõõtudest, nagu `pylinti` skoor ja terminite kogused), et mitmed üliõpilased kasutasid funktsioone kehvasti. Selleks, et funktsioonide kasutust paremaks teha, saaks projekti nõuetesse lisada, et ühte funktsiooni peab vähemalt kaks korda kasutama, sundides üliõpilasi looma kasulikumaid funktsioone.

5.2 Süsteemi piirangud

Nagu iga süsteemiga, kaasnevad ka selle lõputöö raames valminud koodianalüsaatoriga piirangud. Nendeks piiranguteks on:

- süsteem on testitud Pythoni versioonil 3.12 ning töökindlust teistel versioonidel ei saa tagada. Üldiselt on lähestikused Pythoni versioonid üksteisega ühilduvad ning probleeme ei tohiks esineda[37];

[37] https://www.reddit.com/r/Python/comments/16yly4w/comment/k399uk8/?utm_source=share&utm_medium=web3x&utm_name=web3xcss&utm_term=1&utm_content=share_button

- analüüsi sooritamiseks individuaalsele failile, tuleb see tõsta projekti kausta, mille lõpus on `_assignsubmission_file` ning seejärel kokku pakkida arhiivi, millele saaks analüüsi sooritada;
- süsteem ei oska koondhinnet anda projektidele, mis koosnevad mitmest failist. Küll aga oskab see anda kõigile projektifailidele individuaalsed hinnangud. Hetkeseisuga on kursuse „Programmeerimise alused“ iga esitatud projekti kohta üks Pythoni fail (välja arvatud taasesitused) ning probleemi ei tohiks tekkida;
- genereeritud Exceli failis olevad lingid on suunatud päris failidele ning et lingid töötaksid, ei tohi nende failide (projekti failide) asukohti muuta. Küll aga tohib muuta Exceli faili asukohta, kuna lingid on absoluutsed failiteed;
- sisend on ehitatud Moodle'ist allalaaditava projektide struktuuri jaoks. Hetkeseisuga see töötab, aga tulevikus võib projektide struktuur Moodle'is muutuda või võib muutuda platvorm, kus projektid on esitatud. Sellisel juhul praegune lahendus arvatavasti enam ei toimiks ning seda peaks uuendama.

5.3 Võimalikud edasiarendused

Selle süsteemi arendamise käigus tekkisid mitmed ideed, kuidas süsteemi edasi arendada. Järgnevalt on välja toodud mõned edasiarenduse ideed:

- lisada funktsionaalsus individuaalsete failide analüüsiks ning lisada uusi analüüsimismeetodeid. Näiteks oleks võimalik lasta mõnel generatiivsel keelemudelil võrrelda projekti formulatsiooni projekti koodiga, kuna projektid ei ole mahukad. Selleks, et väljund saada, kui palju erines formulatsioon valminud projektist, oleks võimalik kasutada mingit tundeanalüüsi[38] (ingl *sentiment analysis*) masinõppe mudelit;
- luua viis konfigureerida projektide struktuuri. Näiteks kasutades YAML-faili, millega on võimalik kaustade hierarhiaid kirja panna. Seejärel peaks looma süsteemi, mis selle YAML-faili järgi projektid üles leiab;
- luua graafiline kasutajaliides. See võib ühilduda projektide struktuuri konfigureerimisega, kus saaks graafiliselt luua oodatud sisendi struktuuri;

[38] https://akit.cyber.ee/term/13690-sentiment-analysis

- mõned mõõdud on nii-öelda toored, need annavad ainult suhtelist informatsiooni võrreldes teiste projektidega (näiteks Halsteadi aja mõõt ei näita aega, mis kulub algajal koodi kirjutamiseks, aga näitab, kui palju keerukam võis kood olla mõnest teisest programmist). Näiteks üks võimalus on koguda päris andmeid, kui kaua üliõpilastel läheb, et valmis kirjutada mingi projekt ning seejärel võrrelda seda Halsteadi ajaga. Sama saab teha ka keerukuse ja pingutusega;
- arendada mitme faili projektide analüüs. Selle teostamiseks saaks näiteks ühendada mitu projekti faili üheks, asendades `import` laused neile vastavate Pythoni moodulite sisudega. Seejärel saaks analüüsi rakendada samamoodi nagu praeguse süsteemiga;
- tehisintellekti tuvastuse jaoks saaks lasta üliõpilastel järelvalve all kirjutada projektikoode käsitsi. Paralleelselt tuleks luua generatiivsete tehisintellekti mudelitega programme. Viimaks saaks nende andmetega kas puhtalt koodi pealt, või kaasata ka koodianalüsaatori poolt loodud analüüside tulemused, tehisintellekti mudel luua, et tuvastada generatiivse tehisintellekti kasutust projektides.

Need on mõned väga varieeruvate mahtudega ideed, aga mille arendamine on kindlasti kasulik.

6. Kokkuvõte

Kursusel „Programmeerimise alused“ õpetatakse programmeerimist keeles Python. Sellel kursusel on programmeerimise õpetamiseks ka individuaalprojektid, mille eelmise aasta (2024) sügissemestril esitasid 109 üliõpilast. Nende hindamine ning analüüsimine kulutab palju aega, mida õppejõud saaksid pühendada muudele tegevustele.

Selle lõputöö käigus valmis koodianalüsaator, mis suudab automaatselt üles leida kursuse „Programmeerimise alused“ projektide koodid ning neile analüüse sooritada. Süsteem täitis kõik nõuded edukalt: mõnesammuline installeerimine, ühe käsuga analüüs, muudetavad failiteed läbi käivitusargumentide, väljundis on peaaegu kaks korda rohkem mõõte kui oodatud, väljund kirjutatakse Exceli faili, kus iga analüüs on seotud lingiga projekti failiga, 60% lühema tööajaga kui nõutud, süsteem otsib projektikoodid ise üles ning annab hindamisskeemi järgi korrektse hinde.

Valminud süsteem suudab sooritada analüüsi ja hindamise enam kui sajale projektile kahe minutiga. Selle käigus väljastatakse Exceli fail koos suure koguse erinevate analüüsi tulemustega, mis lihtsustab nii hindamist kui analüüsi. Aeg, mis eelnevalt kulus manuaalseks lähtekoodi analüüsiks, on võimalik nüüd pühendada abstraktsemate ideede uurimisele — mõõdud abstraheerivad programmikoodi ära. Tulenevalt analüüsi deterministlikust metoodikast on võimalik võrrelda erinevate aastate vahelisi muutuseid projektide tulemustes. Koodianalüsaatori koodile ligipääs on lisade peatükis 6.

Viited

[1] ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)* (2010), lk 345. DOI: 10.1109/IEEESTD.2010.5733835. http://doi.org/10.1109/IEEESTD.2010.5733835.

[2] Harman M. Why Source Code Analysis and Manipulation Will Always be Important. *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on* (september 2010). Publisher: IEEE, lk 7–9. DOI: 10.1109/SCAM.2010.28. http://doi.org/10.1109/SCAM.2010.28.

[3] Mushtaq Z. ja Rasool G. Multilingual source code analysis: State of the art and challenges. *2015 International Conference on Open Source Systems & Technologies (ICOSST), Open Source Systems & Technologies (ICOSST), 2015 International Conference on* (detsember 2015). Publisher: IEEE, lk 170. DOI: 10.1109/ICOSST.2015.7396422. https://doi.org/10.1109/ICOSST.2015.7396422.

[4] Yang J., Lee Y., Fernandez A. ja Sanchez J. Evaluating and Securing Text-Based Java Code through Static Code Analysis. *Journal of Cybersecurity Education, Research and Practice* 2020.1 (juuni 2020). Publisher: Kennesaw State University ERIC Number: EJ1343162, lk 3. https://eric.ed.gov/?id=EJ1343162.

[5] Tisha S. M., Oregon R. A., Baumgartner G., Alegre F. ja Moreno J. An Automatic Grading System for a High School-level Computational Thinking Course. *2022 IEEE/ACM 4th International Workshop on Software Engineering Education for the Next Generation (SEENG), Software Engineering Education for the Next Generation (SEENG), 2022 IEEE/ACM 4th International Workshop on, SEENG* (mai 2022). Publisher: ACM, lk 20–25. DOI: 10.1145/3528231.3528357. http://doi.org/10.1145/3528231.3528357.

[6] SELIUTIN D. ja YASHYNA E. INTELLECTUAL CODE ANALYSIS IN AUTOMATION GRADING. *Radioelectronic & Computer Systems / Radìoelektronnì ì Komp'ûternì Sistemi* 4 (oktoober 2024). Publisher: National Aerospace University named after M.E. Zhukovsky "Kharkiv Aviation Institute", lk 68–82. DOI: 10.32620/reks.2024.4.06. http://doi.org/10.32620/reks.2024.4.06.

[7] Orr J. W. ja Russell N. Automatic Assessment of the Design Quality of Python Programs with Personalized Feedback. Tehniline raport. ERIC Number: ED615527. International Educational Data Mining Society, 2021, lk 495–501. https://eric.ed.gov/?id=ED615527.

[8] Jaggo J. JavaScripti staatilise koodianalüsaatori loomine teenusena. Tehniline raport. Tartu: Tartu Ülikool Arvutiteaduse instituut, 2013, lk 4. https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=29311.

[9] Hunt A. The pragmatic programmer : from journeyman to master. Reading, Mass. : Addison-Wesley, 2000. http://archive.org/details/isbn_9780201616224 (18.04.2025).

[10] Puulmann K.-A. Python module for automatic testing of programming assignments. Tehniline raport. Tartu: Tartu Ülikool Arvutiteaduse instituut, 2014, lk 1–40. https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=40971.

[11] Kohn T. ja Staub J. The Two Powers: How Pascal and Python Shaped Programming Education. *Informatics in Education* 23.4 (detsember 2024). Publisher: Vilnius University, lk 837–842. DOI: 10.15388/infedu.2024.30. https://doi.org/10.15388/infedu.2024.30.

[12] ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)* (2010), lk 223. DOI: 10.1109/IEEESTD.2010.5733835. http://doi.org/10.1109/IEEESTD.2010.5733835.

Lisad

Link koodianalüsaatori lähtekoodile: https://kodu.ut.ee/~merilipi/koodianalysaator.zip

Lähtekoodi varukoopia: https://drive.google.com/drive/folders/1VqJEm7Jw2J5NpI3HNo4u2jBEwhh9EJng

Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Merili Pihlak,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose „Koodianalüsaatori arendus kursuse „Programmeerimise alused“ projektide tehnilise analüüsi automatiseerimiseks“, mille juhendaja on Reelika Suviste, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada Tartu Ülikooli digitaalarhiivi kuni autoriõiguse kehtivuse lõppemiseni;
2. annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi kaudu Creative Commonsi litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni;
3. olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile;
4. kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Merili Pihlak

15.05.2025