TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Kaarel Loide

Pythoni süntaksivigade analüüs algajasõbralikumate veateadete kuvamiseks

Bakalaureusetöö (9 EAP)

Juhendaja: Aivar Annamaa
Juhendaja: Vesal Vojdani

Tartu 2020

Pythoni süntaksivigade analüüs algajasõbralikumate veateadete kuvamiseks

Lühikokkuvõte:
Bakalaureusetöö eesmärk on uurida varasemate uurimuste põhjal levinumaid algajate poolt tehtud süntaksivigasid programmeerimisekeeles Python ning analüüsida ja võrrelda erinevaid lahendusi, mille abil püütakse selgemaid veateateid luua. Omandatud teadmisi silmas pidades luuakse töö raames tarkvara, mis pakub parendatud veateateid algajate seas enim levinud süntaksivigade kohta.

Võtmesõnad:
Python, koodianalüüs, veateated, programmeerimine, süntaksivead

CERCS: P175 Informaatika, süsteemiteoooria

Python syntax error analysis for displaying beginner friendly error messages

Abstract:
The goal of this bachelor's thesis is to review literature on the most common syntax errors made by beginners in the programming language Python and analyse different solutions for making the errors more understandable to them. Using the acquired knowledge, a software solution will be created. The aim of this software is to offer improved error messages for the most common syntax errors made by beginners.

Keywords:
Python, code analysis, error messages, programming, syntax error

CERCS: P175 Informatics, systems theory

Sisukord

1 Sissejuhatus

On tehtud mitmeid uuringuid kõige levinumate vigade kohta erinevates programmeerimiskeeltes ja uuritud, milliseid veateateid erinevate vigade puhul kompilaator tagastab. Enamasti on leitud, et algajate poolt tehtud vead on sageli väga lihtsakoelised ja kogenenuma programmeerija poolt väga kergesti parandatavad. Probleemiks on aga kompilaatorite poolt pakutavates veateadetes kasutusel olevad erikeelsed väljendid, mis algajate jaoks jäävad arusaamatuks ja võivad aitamise asemel hoopis segadust tekitada. Lisaks veateadetes kasutatavale žargoonile eksivad kompilaatorid tihti ka vea asukohale viitamisega, sest viga tuvastatakse mitu sümbolit või isegi rida hiljem. Kuna viga koheselt ei tuvastata, siis on hiljem ka raske selgitada, millest see on tingitud ja seetõttu tekitavad ka eksitavad asukohaviited [1, 2, 3]. Samuti on uuritud, kui palju veateateid loetakse ja kui lihtne on neist aru saada [4, 5].

Käesoleva bakalaureusetöö raames uuritakse läbi varasemalt teostatud teadustööde, mis on kõige sagedasemad süntaksivead programmeerimisekeeles Python. Läbi töötatud allikate põhjal luuakse täpsemad ja algajasõbralikumad veateated ning tarkvara prototüüp vigade tuvastadamiseks. Lõpptulemusena avaldatakse prototüüp Pythoni paketirepositooriumis PyPi ja integreeritakse algajatele suunatud arenduskeskkonda Thonny.

Eesmärk on luua universaalne veateadete parendaja, mis on laiendatav ja mida on võimalik kasutada Pythonis kirjutatud tööriistades. Lisaks praktilisele poolele on plaanis omandada parem arusaam käesolevast probleemist ja sellega kaasnevatest nüanssidest.

Kuigi algajate vigade uurimisel käsitletakse tavaliselt erinevaid veatüüpe, siis selle töö raames keskendutakse süntaksivigadele. Sellel kitsendusel on mitmeid põhjuseid. Erinevad uuringud on leidnud, et algajate seas on just süntaksivead kõige levinumad [3, 4], lisaks on Python äärmiselt napisõnaline seda tüüpi vigadest teatamisel. Need kaks asjaolu koos teevad just süntaksivead oluliseks valukohaks algajatele. Täiendavaks asjaoluks on ka see, et süntaksivigu saab vaadelda täiesti eraldiseisvana teistest veatüüpidest, sest neid tuvastab kompilaator enne programmi käivitamist. Seetõttu saab hästi keskenduda ainult süntaksile, ja samal ajal ei pea muretsema, et muud veatüübid segama hakkaksid.

Prototüübi valideerimiseks kasutatakse arenduse käigus loodud üksusteste erinevate süntaksivigade kohta. Lisaks testitakse lõppfaasis olevat prototüüpi Wong et al. poolt koostatud andmestikust [6] välja otsitud vigaste koodijuppide peal, mis vastavad käesoleva töö skoobile.

Edasine töö on üles ehitatud järgmiselt. Peatükis 2 antakse lühike ülevaade käesolevale tööle relevantsetest Pythoni aspektidest, ja tutvustatakse arenduskeskkonda Thonny, ning põhjendatakse, miks just Thonny võiks olla esimene loodava teegi kasutaja. Lisaks võetakse kokku varasemad tööd, mis on tehtud veateadete ja nende parandamise kohta, kirjeldatakse täpsemalt erinevaid veatüüpe ning põhjendatakse otsust keskenduda süntaksivigadele.

Peatükis 3 arutletakse varasemalt tehtud tööde ja lahenduste üle ja võrreldakse, miks käesolevas töös loodud prototüüp on vajalik ja praegustest lahendustest erinev.

Prototüübi ülesehitust on lähemalt selgitatud peatükis 4, kus võetakse kokku prototüübi läbiv idee ning käesoleva töö raames olulised nõuded ja kitsendused. Seejärel kirjeldatakse põhjalikumalt tarkvara ülesehitust ja olulisemaid valikuid, mis arenduse käigus tehti. Lõpuks tuuakse näiteid loodud rakenduse kasutamisest ning loetletakse üles loodud vigade kontrollid koos kirjeldustega.

Lõpuks arutletakse 5. peatükis saavutatud tulemuste üle läbi kogemuste ja valideerimise tulemuste ning tuuakse välja võimalikud edasiarendused.

2 Taust

Järgnevas peatükis on juttu sellest, miks Python on sobiv kandidaat algajasõbralikumate veateadete jaoks ja miks Thonny on käesoleva töö jaoks sobilik arenduskeskkond millega integreeruda. Lisaks arutletakse varasemate uurimuste üle veateadete kasulikkuse ja loetavuse teemal ning tehakse nende põhjal järeldusi, mida hiljem tarkvara loomisel arvesse võtta.

2.1 Python

Python on laialdaselt levinud üldotstarbeline interpreteeritud keel. Pythoni koodi jooksutamine on oluliselt lihtsam võrreldes kompileeritavate keeltega nagu Java ja C++. Interpreteeritud keeltele omaselt pole Pythoni programmikoodi vaja enne käivitamist vahesammuna baitkoodi kompileerida – see tehakse kasutajale nähtamatult [7]. Seepärast on esimeste tulemuste nägemiseks vaja programmikood valmis kirjutada ja seejärel käivitada. See on oluline lihtsustus alustavale programmeerijale, sest eemaldab ühe tehnilise sammu, mis võib algajale eemaletõukav ja hirmuäratav olla.

Veel üks Pythoni populaarsuse põhjus on asjaolu, et keele süntaksi võrreldakse sageli pseudokoodiga, mis viitab sellele, et tegemist on väga lihtsasti mõistetava ja selge programmeerimiskeelega. Selgus tuleneb vähesest keelele omasest žargoonist, mille tõttu on programmikood ka tihti palju lühem võrreldes teiste üldotstarbeliste programmeerimiskeeltega. Samuti on paljud võtmesõnad väga lihtsad ning kohati on programmikood väga sarnane tavaliste ingliskeelsete lausetega.

Koodi struktuuri loomisel kasutatakse Pythonis sulgude asemel tühikuid, tabulaatoreid ja reavahetusi. See loob programmeerija jaoks visuaalselt palju loetavama terviku ja seab oluliselt rangema malli koodi stiilile. Olenemata sellest, et Pythonit nimetatakse skriptimiskeeleks, on programmeerimiskeeles olemas kõik objektorienteeritud programmeerimise võimalused. See teeb Pythoni tänu kasutajasõbralikkusele väga heaks alguspunktiks objektorienteeritud võtete õppimisel, kuna paljud teised levinud objektorienteeritud keeled vajavad lisaks objektorienteeritusele ka keeruka süntaksiga toime tulemist [7].

2.2 Thonny

Thonny [8] on integreeritud arenduskeskkond, mis on loodud selleks, et toetada õpilasi Pythoni õppimisel. Lisaks on olemas funktsionaalsus, mis tuleb kasuks ka õpetamisel.

Algajaid toetavaid funktsioone on palju. Lihtsamad neist on süntaksi-põhine koodi värvimine ja alati avatud käsurea liides. Need on aga olemas ka enamikes teistes arenduskeskkondades nagu näiteks IntelliJ ja Eclipse. Thonny spetsiifilisem abivahend on näiteks silur, mis võimaldab sammhaaval lähtekoodi läbi mängida, samal ajal visualiseerides olulisi osasid koodist. Lisaks visualiseerib Thonny kasutaja jaoks igal siluri sammul ka

hetkel deklareeritud muutujad ja nende väärtused. Olemas on ka *Assistant* ehk abiline, mis pakub vastavalt lähtekoodile erinevaid soovitusi ja hoiatusi võimalike vigade kohta.

Lisaks õpilastele toetab Thonny ka õppejõudusid ja õpetajaid, salvestades vajadusel iga kasutaja tegevuse logi failidesse. Selle abil on oluliselt lihtsam läbi viia erinevaid töösid, kuna logid lihtsustavad hiljem hindamisprotsessi ja aitavad veenduda, et iga õpilane on oma töö teinud iseseisvalt. Logid võimaldavad ka hiljem läbi viia erinevaid analüüsivaid uurimusi kõikvõimalike õpilaste käitumismustrite, arenguprotessside ja tehtud vigade kohta. Thonny toetab nii Windowsi, MacOSi, kui ka Linuxit, kaasa arvatud Raspbiani. Tänu sellele on arenduskeskkond sobilik lahendus kõikvõimalikeks õppeainetes, alustades sissejuhatavatest programmeerimiskursusest, kuni erinevate robootikakursusteni välja.

Kuna Thonny on loodud Pythoniga ja selle peamised kasutajad on algajad, siis on see ideaalne keskkond, milles rakendada töö raames loodud teeki, sest loodud veateated on samuti mõeldud algajatele ja teek ise on suunatud just Pythoni-põhistele tööriistadele.

2.3 Levinumad veatüübid

Pythoni puhul on Pritchard välja selgitanud, et kõige sagedasemad veateadete tüübid on *SyntaxError*, *NameError*, *IndentationError* ja *EOFError* [2]. Sarnase uurimuse on läbi viinud Anne Kelley, kes jõudis mõnevõrra erinevate tulemusteni [3]. Massachusetts Institute of Technology ülikoolis läbi viidud uuringu käigus leidis ta, et enim levinud veatüübid on hoopis *TypeError*, *AttributeError*, *NameError* ja *SyntaxError*.

Uurimused langesid kokku kahe veatüübi osas, *NameError* ja *SyntaxError*. Mõlemad viitavad reeglina kas trükiveale, mõnele puuduvale sümbolile või üleliigsele tühikule. Mõningase kogemusega inimese jaoks on selliste vigade märkamine lihtne ja sageli ka ebaoluline, sest kasutusel olev arenduskeskkond oskab neid tuvastada juba enne koodi käivitamist. Samu vigu on värskel programmeerijal oluliselt raskem leida, sest reeglina ei soovitata kohe kasutada keerukaid arenduskeskkondasid, mis taolisi vigu ise parandavad, kuna need võivad olla oma tohutute võimalustega hirmuäratavad ja segadust tekitavad. Lisaks on alguses hea kasvatada harjumust ise koodi kirjutada. Mainitud veateated sisaldavad ka erikeelseid väljendid nagu näiteks *syntax* ja *EOF*, mis koos vähese selgituse ja vale asukoha viidaga teevad vea lahendamise ebavajalikult keeruliseks.

Kuna uurimused [3, 2] langesid kokku *NameError* ja *SyntaxError* puhul, siis vaadeldakse selles töös neid. *NameError* on käitusviga. Käitusvead avastab Python alles siis, kui interpretaator on problemaatilise instruktsioonini jõudnud. *NameError* tähendab, et üritati välja kutsuda funktsiooni või kasutada muutuja nime, aga seda pole hetkeseisuga defineeritud. Reeglina on *NameError* tingitud puuduvatest jutumärkidest sõnena mõeldud koodijupi ümber või trükiveast muutuja nimes [9]. *SyntaxError* on veatüüp, mille tähendus võib pealtnäha tunduda üsna ilmne. Nimelt viga on tehtud süntaksis ja kirjutatud kood ei ole korrektne Pythoni programmikood. Seda tüüpi viga suudab Python tuvastada juba enne koodi käivitamist [9]. Kuigi võib tunduda, et süntaksivead on lihtsasti parandatavad, siis tegelikkuses on tegemist väga laia kategooriaga, mis hõlmab suurt hulka erinevaid vigasid. Süntaksivigade puhul on sageli ka probleemiks vea tuvastamise asukoht ja napi selgitusega veateated, sellest tulenevalt on need eriti problemaatilised algajate seas. Eksitavad kohaviidad ja puudulikud veateated võivad vähem kogenenud programmeerijas tekitada segadust ja seetõttu võidakse teha vea parandamise asemel hoopis vigu juurde või lähenetakse täiesti vale nurga alt, mille tulemusel programmikood küll käivitub, aga tegeliku vea põhjuseni ei jõuta [10]. Võttes arvesse, et mõlemad eelnevalt mainitud uurimused tõid välja, et süntaksivead on algajate seas väga levinud [3, 2] ning selle, et antud veatüübi teated on kohati väga ebaselged ja segadusse ajavad, siis käesoleva töö raames keskendutaksegi programmeerimiskeele Python *SyntaxError* tüüpi vigade tuvastamisele ning vastavate veateadete täiustamisele läbi koodianalüüsi. Kuna *IndentationError* on *SyntaxErrori* alamklass, siis tehakse katsetusi ka selle vea tuvastamises ja veateate täiendamises.

2.4 Veateadete parandamisest

Veateadete parandamine ja täiustamine on üpris levinud teema programmeerimiskeeltega seotud teadustöödes. Järgnevas jaotises võetakse kokku mõningad relevantsemad uuringud järgmistel teemadel: kas veateateid üldse loetakse, millised on head veateated ja kas veateadete täiustamisest on kasu.

Barik et al. viisid läbi uuringu [4] bakalaureuse ja magistri taseme õpilaste seas. Nad jälgisid osalejate silmade tööd erinevate vigaste koodijuppide parandamisel, et selgitada välja, kas ja kuidas programmeerijad neile kuvatavaid veateateid kasutavad. Uuringu käigus tuli välja, et veateateid tõepoolest loetakse. Selgus, et nende lugemisele võib kuluda kuni 25% arenduskeskkonnas oldud ajast ja teadetest aru saamine võib osutuda sama keeruliseks kui lähtekoodi mõistmine.

Mitmetes teadustöödes on uuritud, kas veateadete täiendamine aitab programmeerijaid. Lisaks on uuritud, millisel kujul veateated on kõige tõhusamad. [11] Täiendatud veateadete efektiivsuse kohta on tihti jõutud vastuolulisetele järeldustele. Becker et al. on välja toonud mitmeid varasemaid töid [12], milles on jõutud nii positiivsete kui ka negatiivsete tulemusteni. Nad tõid välja, et uuringud on oma metoodikalt küll sarnased, kuid tulemusi mõõdetakse väga erinevalt, ja tänu sellele on järeldused ka vastuolulised. Viies läbi täiendavaid katseid, jõuti tulemusteni, mis toetasid mõlemat poolt argumendist. Pakuti välja, et täiendatud veateated siiski vähendavad tehtavate vigade arvu, aga ei paranda otseselt testide tulemusi. Põhjuseks toodi asjaolu, et uuringutes kasutatud uute teadete efekt on piisavalt väike, et mitte kajastuda õpilaste testide tulemustes. Lõpetuseks pakuti välja, et ehk tuleks edasistes uurimustes hoopis rohkem keskenduda täiendatud teadete disainimisele.

Barik et al. tõid oma töös [4] välja kaks olulist faktorit, miks veateadete lugemine nii koormav ja aeganõudev tegevus on. Esiteks on veateadetes segamini tava- ja erikeelsed sõnad, mille tõttu peab programmeerija pidevalt konteksti vahetama. Näiteks veateates *SyntaxError: EOL while scanning string literal* on vaheldumisi erikeelsed väljendid nagu *EOL*, *string* ja *literal* argikeelsete sõnadega *while* ja *scanning*. Teiseks on veateated tavaliselt nii napisõnalised, et neist aru saamiseks tuleb pidevalt vahetada fookust lähtekoodi ja teate teksti vahel. Sarnaselt esimesele asjaolule hajutab ka see arendaja tähelepanu ja pideva ümberlülitamise tõttu võtab ebavajalikult palju aega ja energiat.

Kirjeldatud tulemuse põhjal võiks arvata, et kasuks tuleksid visuaalsemad veateated, mis tooksid juba sõnumi kehas välja vigase koha koodist, et programmeerija ei peaks tegema oma peas kulukaid ümberlülitusi. Nienaltowski et al. viis õpilaste seas läbi küsitluse [11], kus uuriti, kas visuaalsemad veateated aitavad vigadest paremini aru saada. Uuringus kasutati kolme tüüpi teateid: lühivormis sõnum, pikas vormis sõnum ja visuaalne teade. Nendest kolmest saavutas kõige kehvema tulemuse visuaalset tüüpi teade. Leiti veel, et rohkem informatsiooni veateates ei tähenda, et õpilased jõuaksid korrektse tulemuseni kiiremini või tihedamini. Lisatud informatsioon ei pruugi algajatele midagi pakkuda, kuna on tihti liiga tehniline.

Pikemad või visuaalsemad veateated ei ole tingimata paremad, aga siiski on leitud, et veateadete täiendamisest on kasu [13, 5]. Näiteks viis Becker läbi uuringu [13], kus kasutas spetsiaalselt välja arendatud veateateid täiustavat arenduskeskkonda Decaf, mida kasutades õpilased pidid lahendama programmeerimisülesandeid. Õpilastel paluti hiljem ise kirjeldada enda kogemust ja suur osa leidis, et Decaf ja selle poolt pakutud täiendatud veateated meeldivad neile, ning teevad programmeerimise õppimist lihtsamaks. Prather et al. jõudsid sarnasele tulemusele [5], kus pärast katseid küsiti õpilaste käest arvamust täiendatud veateadete kohta. Õpilased, kes sattusid ülesannetega hätta väljendasid, et nende arvates oli parematest teadetest palju kasu ja tänu neile jõuti ka lõpuks õige lahenduseni. Oli ka osalisi, kelle jaoks tekitasid teated pigem segadust, aga see oli tõenäoliselt tingitud õpilaste üldisest tasemest ja asjaolust, et teema oli nende jaoks nii võõras, et lisainformatsioonist polnud mingit kasu.

Becker tõi välja kolm potentsiaalset viisi, kuidas keerukate veateadete probleem võiks lahenduse leida [13]. Ta pakkus, et lahendus tuleks kas keele disaineritelt, kompilaatorite disaineritelt või tööriistade loojatelt. Käesoleva töö raames lähenetakse probleemile tööriistade loojate poolelt ja uute veateadete puhul on võetud arvesse varasemate uurimuste käigus välja toodud asjaolusid, et visuaalsed veateated ei ole kuigi hea lahendus [11] ning seda, et eri- ja argikeelsete sõnade segamini kasutus on takistavaks asjaoluks veateadete mõistmisel [4].

3 Praegused lahendused

Paremate veateadete loomiseks Pythonis on mitmeid lahendusi, kuid vähesed neist on suunatud algajatele. Selles peatükis keskendutakse kahe lahenduse kirjeldamisele, mis on käesoleva töö jaoks relevantsed, sest keskenduvad just vigade arusaadavamaks muutmisele algajate jaoks.

3.1 TigerJython

TigerJython [14] on alternatiivne arenduskeskkond Pythoni jaoks, mis sisaldab endas spetsiaalselt välja töötatud kompilaatorit, mis on kirjutatud programmeerimiskeeles Scala. TigerJython on sarnaselt Thonnyle suunatud algajatele ja mõeldud Pythoni õpetamiseks. Tegemist on küll projektiga, mille üks eesmärk on edastada algajasõbralikumaid veateateid, aga seda tehakse läbi eraldi arenduskeskkonna. TigerJythonis kasutusel olev parser on küll vabavaraliselt saadaval [15], aga see pakub hetkel liidest ainult JavaScripti põhistele tarkvaradele ja selle tõttu ei ole antud lahendust võimalik Pythoni-põhistes tarkvarades nagu Thonny kasutada.

3.2 Friendly Traceback

Friendly Traceback [16] on Pythoni teek, mis pakub algajatele suunatud veateateid. Teek laiendab olemasolevaid Pythoni veateateid, lisades neile põhjalikud selgitused vea kohta ja tuues visuaalse viite koodile, mis vea põhjustas. See tarkvara on eesmärgilt üsna sarnane käesoleva töö raames loodud prototüübile, kuid keskendub rohkem sellele, et tõlkida vigu arendaja emakeelde. Teek on ka kasutuses arenduskeskkonnas Thonny, kus on seadetest võimalik sisse lülitada erinevaid abistavaid tööriistu, millest üks on Friendly Traceback. Thonny puhul jõuavad teegi poolt genereeritud teated konsooli tavalise veateate asemele. Selle bakalaureusetöö raames loodud tarkvara siht on aga luua soovituslikke selgitusi tuvastatud vigade puhul ja pakkuda neid eraldiseisvalt tavapärasest veateatest (Thonny puhul näiteks *Assistant* akna kaudu). Erinevalt Friendly Tracebackile keskendub selle töö raames loodud tarkvara just süntaksivigadele. Lisaks on eesmärk luua võimalikult paindlik ja arendajasõbralik tarkvara, pakkudes võimalust igal arendajal soovi korral lisada uusi kontrolle ja täiendatud veateateid, endale ja oma projektile sobival kujul.

Mõlemad eelnevalt mainitud lahendused on küll suunatud algajatele, kuid neil on siiski puuduseid just veateadete lihtsustamise osas. Mõlemad pakuvad võimalust veateateid tõlkida toetatud keeltesse, aga väga palju rõhku pole pandud sellele, et veateated ise kasutaksid võimalikult lihtsat sõnavara ja oleksid ülesehituselt algajasõbralikumad.

3.3 Veel sarnaseid lahendusi

Lisaks eelnevalt kirjeldatud lahendustele leidub veel sarnaseid lahendusi [3, 13], need aga erinevad ühe väga olulise asja poolest. Tegemist on lahendustega, mis töö autorile teadaolevalt ei ole avaliku lähtekoodiga ega vabalt kättesaadavad ja kasutatavad mistahes projektides. Kelley poolt loodud tarkvara [3] paistab disaini kirjelduse järgi üsna sarnane käesoleva töö raames loodud tarkvarale, kuid on loodud eesmärgiga erinevaid Pythoni vigasid võimalikult täpselt klassifitseerida. Rõhku on pandud just õigete veatüüpide tuvastamisele, mitte veateadete sisule. Becker on samuti loonud enda uurimuse [13] jaoks arenduskeskkonna, mis üritab veateateid täiendada. Selle projekti puhul on aga tegemist Java keelele suunatud lahendusega. Neid asjaolusid arvestades ei saa kahjuks antud projekte kuigi hästi võrrelda käesoleva tööga, kuid need olid siiski piisavalt sarnased, et välja tuua.

4 Valminud tarkvara

Järgnev peatükk on jagatud alamosadeks, kus esmalt kirjeldatakse tarkvara prototüübi üldist ideed ning käesoleva töö raames olulisi nõudeid ja kitsendusi. Seejärel süvenetakse põhjalikumalt programmi ülesehitusse ja seletatakse lahti olulisemad valikud tarkvara disainimisel. Lisaks tuuakse näiteid, kuidas prototüüpi kasutada ning kuidas selle funktsionaalsust iga kasutaja ise laiendada saab. Lõpetuseks kirjeldatakse Thonny jaoks loodud pistikprogrammi ja kontrolle, mis on prototüübis implemeteeritud.

4.1 Idee

Käesoleva lõputöö raames valminud tarkvara on eelkõige suunatud erinevate tööriistade loojatele, kes kasutavad arenduseks Pythonit ja soovivad enda kasutajatele mingil moel täiendatud või muudetud veateateid edasi anda. Programm on disainitud nii, et isegi siis, kui arendaja ei vaja sisse ehitatud süntaksivigade kontrolle, on tal võimalik vähese vaevaga ise uusi kontrolle lisada ja olemasolevaid eemaldada.

Tarkvara on loodud eesmärgiga olla võimalikult paindlik, aga baasimplementatsioonis on keskendutud Pythoni koodis esinevate süntaksivigade kontrollidele, täpsemalt sellistele süntaksivigadele, mis on algajate seas enim levinud. Tarkvaraga kaasnevad mitmed kontrollid erinevate süntaksivigade kohta. Igale kontrollile vastab üks või rohkem täiendatud veateadet, mille ülesehitusel on silmas peetud varasematest uurimustest [4, 11] selgunud järgnevaid asjaolusid: pikad ja põhjalikud veateated ei pruugi alati kasulikud olla ning veateadetes võiks kasutada võimalikult vähe erikeelseid väljendeid.

4.2 Nõuded ja kitsendused

Juba arenduse algfaasis sai prototüübi üheks põhiliseks eesmärgiks paindlikkus, peamiselt seetõttu, et kontrollide loomine ja lisamine muutus kiiresti väga segaseks ja raskesti hallatavaks. Lisaks oli motivaatoriks pakkuda jätkusuutlikku lahendust, mida on edaspidi võimalik kergesti edasi arendada või siis kasutajatel ise vastavalt vajadusele laiendada. Siit kujunes välja nõue, et uue kontrolli lisamise ainuke keerukus on kontroll-loogika ja veateate väljamõtlemine ning kirja panemine. Teek ise pakub lahenduse uue loogika lisamiseks olemasolevasse süsteemi.

Teine nõue tuleneb varasematest uurimustest ja nende põhjal tehtud järeldustest. Nõue seisneb selles, et teegi poolt kaasa tulevad veateated sisaldavad minimaalselt erikeelseid väljendeid ja on võimalikult lühikesed, aga samas piisavalt informatiivsed.

Kuna programmist pole suurt abi, kui seda keegi ei kasuta, siis kolmandaks nõudeks on, et teeki oleks võimalik kasutusele võtta mõnes olemasolevas tööriistas. Huvide ja sihtrühma kokkulangemise tõttu osutus selliseks tööriistaks arenduskeskkond Thonny.

Vigade tuvastamine ja veateadete loomine on äärmiselt lai teema isegi ühe programmeerimiskeele raames. Seda arvesse võttes on selle töö käigus valminud prototüübile

seatud ka mõningad kitsendused, et skoopi vähendada. Teegi põhifunktsionaalsusel endal olulisi kitsendusi ei ole, kuid loodud süntaksivigade kontrollidel siiski mõningad on.

Esimeseks suuremaks kitsenduseks on see, et erilist tähelepanu ei pöörata jõudlusele, kuna kontrollid on suunatud eelkõige algajatele ja nende loodud programmikood on reeglina üpris lühike. Samal põhjusel ei arvestatud erinevate Pythoni funktsionaalsustega, mis on suunatud pigem edasijõudnud kasutajatele. Näiteks erinevad keerukad operatsioonid nimekirjadega, tüübivihjed, dekoraatorid ja muu taoline. Eeldatud on ka seda, et lühikese koodi puhul on vigade arv väike või sama viga esineb korduvalt. Seetõttu pole väga süvitsi toetatud erinevad stsenaariumid, kus on mitmed erinevat sorti vead ühes koodijupis omavahel kombineeritud. Teek küll toetab mitme vea leidmist ühest failist, aga analüüsi ülesehituse tõttu tagastatakse reeglina esimesena leitud ja teegi arvates olulisim veateade. Mitme veateate tagastamine saab juhtuda ainult olukorras, kus tuvastatava vea kontrollidele on määratud sama tase või koodis on mitu sama tüüpi viga.

4.3 Valikud

Esimene suurem disainiotsus oli valik kaasata Pythoni teek Parso [17], mis erinevalt Pythoni enda *ast* moodulist võimaldab ka vigase koodi puhul luua pooliku süntaksipuu. Tänu Parsole on mitmete vigade tuvastamine oluliselt lihtsustatud, sest teegi poolt loodud süntaksipuust on hiljem võimalik väga kerge vaevaga vigasid sisaldavaid tippe kätte saada ja neid edasi analüüsida. Kuigi Parso oli suureks abiks teatud vigade puhul, siis tuli ka välja väga palju olukordi, mille puhul Parso ei suutnud vigast koodi tuvastada või teegi poolt pakutud vea asukoht oli tegelikkusest erinev. Lisaks mainitud puudustele andis Parso ka mõnes situatsioonis tuvastatud vea kohta mittetäielikku infot, lõigates vigase koodijupi liiga vara ära. Neid puudujääke arvesse võttes tuli kasutada ka alternatiivseid lähenemisi, näiteks lähtekoodi rida haaval läbi töötamine ja vigase rea Pythoni tokeniteks tegemine.

Üsna kiiresti selgus, et veateadete jaoks oleks vaja üles ehitada mingi lihtsasti hallatav süsteem, mille abil oleks võimalik samale veateatele mitmes kohas viidata ja vastavat teadet ka kasutada. Esimene vajadus sellise süsteemi järgi tekkis üksusteste tehes. Oli vaja veenduda, et iga loodud kontroll tagastaks õige teate. See oleks aga kiiresti üle pea kasvanud, sest alguses muutusid veateated pidevalt ja kui neid oleks pidanud mitmes kohas korduvalt muutma, siis see oleks arendust oluliselt aeglustanud ja uute teadete lisamist kordades raskendanud. Lõpuks valmis veateadete jaoks süsteem, kus igale teatele vastab mingi unikaalne inimloetav kood, mille järgi on veateadet ennast võimalik sõnastikust küsida. Lisaks tuli teadetele lisada tugi muutujate jaoks, et toetada näiteks vea asukohale viitamist rea numbriga. Sarnaselt kontrollmeetoditele on ka veateateid võimalik lisada, muuta ja eemaldada. Veateadete sõnastik koos selle haldamiseks loodud meetoditega asub koodirepositooriumis (Lisa I) failis *error_explainer/messages.py*.

Järgmine suurem otsus oli toetada kahte tüüpi kontrollmeetodeid, mille käivitamiseks on erinevad tingimused. Esiteks on olemas *check* tüüpi meetodid. Need on kontrollid, mis

käivitatakse ainult siis, kui kontrollitav programm ei kompileeru. Baasprogrammiga kaasa tulevad kontrollid on põhiliselt just seda tüüpi, sest süntaktiliselt ebakorrektne kood ei kompileeru ja just süntaksivigadele on selles töös keskendutud. Teiseks kontrollitüübiks on *force_check* kontrollid. Seda tüüpi kontrolle käivitatakse ka siis, kui programmikood kompileerub. Need kontrollid on mõeldud selleks, et oleks võimalik tuvastada ka selliseid vigu, mis otseselt programmi kompileerimist ei takista. Olemasolevatest kontrollidest on selliseks näiteks taanete kontroll, sest vead taanetes ei pruugi alati olla piisavad, et takistada koodi käivitamist.

Lisaks erinevatele käivitustingimustele on kontrollmeetodeid võimalik ka grupeerida tasemetesse 0-99. Vajadus erinevateks tasemeteks tuleneb sellest, et esineb kontrolle, mida ei ole mõtet käivitada kui mingi teine viga on juba leitud ja seetõttu on tasemed üles ehitatud nii, et olenemata kontrollmeetodi tüübist käivitatakse kõik ühe taseme meetodid enne järgmisele edasi liikumist. Süsteem peatab järgnevate tasemete käivitamise esimese vea leidmisel.

Näiteks, kui joonisel 1 kujutatud vigase koodi puhul asuksid kõik kontrollid ühel tasemel, siis annaks analüsaator vea nii puuduva jutumärgi kui ka puuduva kooloni kohta, kuigi tegelikult on ainuke viga puuduv lõpetav jutumärk.

```
if x == "foo:
    pass
```

Joonis 1. Näide tasemete vajadusest

Kuna esimestes iteratsioonides oli prototüübi koodibaas üsna segane ja uute kontrollide lisamine muutus iga korraga järjest keerulisemaks, siis tuli leida mingi lahendus kontrollmeetodite organiseerimiseks ja haldamiseks. Sobivaks lahenduseks osutus dekoraator-mustri [18] kasutusele võtmine.

Pythoni dekoraator-muster on süntaktiline abivahend, mis võimaldab loetavamalt kirjeldada meetodeid, mis töötlevad teisi funktsioone. Selle töö kontekstis on mustrit kasutatud, et registreerida meetodeid kontrollmeetodite sõnastikku. Kuna annotatsiooniga oli vaja kaasa anda parameetreid, siis tuli realiseerida keerukam struktuur, mida on näidatud joonisel 2.

Loodud meetodi abil on uue kontrolli lisamiseks tarvis kirja panna viga tuvastava meetodi loogika ja seejärel märgistada *@add_check* annotatsiooniga, mis tähistab, et kontroll on vaja käivitada iga kord, kui põhimeetodit *run_checks* välja kutsutakse.

```
checks = defaultdict(dict)
force_checks = defaultdict(dict)

def add_check(force: bool, level=99) -> Callable:
  if not 0 <= level <= 99:
      raise AttributeError("Level should be between 0 and 99")

  def dec(func):
      if type(force) != bool:
          raise AttributeError("Force parameter must be defined in
              the decorator")
      if force:
          force_checks[level][func.__name__] = Check(level, func)
      else:
          checks[level][func.__name__] = Check(level, func)

      def wrapper(**kwargs):
          return func(*kwargs)

      return wrapper

  return dec
```

Joonis 2. Kontrollide lisamiseks loodud dekoraator meetod

4.4 Ülesehitus

Programm on üles ehitatud eelnevalt defineeritud kontrollidele, mis kontrollmeetodi väljakutsel kindlas järjekorras läbi jooksutatakse. Kontrollid on mõeldud mingi kindla vea tuvastamiseks ja sellepärast on need kõik veidi erinevate lähenemiste ja sõltuvustega. Mõned kontrollid kasutavad näiteks nii vea tuvastamiseks kui ka vea asukoha määramiseks varasemalt välja töötatud lahendust Pythoni teegi Parso näol. Teised aga lähenevad probleemile tervet faili ridahaaval läbi töötamise kaudu.

Kuna teek on mõeldud tööriistade arendajatele, siis on eeldatud, et kõigil ei ole samasugused vajadused kontrollide osas. Seda silmas pidades on tarkvara disainitud nii, et kaasa tulevaid kontrolle on võimalik välja lülitada ja uusi kontrolle on ka ise võimalik juurde lisada. Arenduse käigus selgus hulgaliselt operatisoone, mis tulevad kasuks praktiliselt iga vea tuvastamisel. Sellised operatsioonid on eraldatud utiliit-meetoditeks, et kergendada tulevikus uute kontrollide lisamist nii teegi haldajal kui ka kasutajatel, kes otsustavad enda projektile spetsiifilisi kontrolle luua. Mõned näited sellistest meetoditest on näiteks faili sisse lugemine ridade kaupa või kogu tekstina, tekstirea tükeldamine Pythoni lekseemideks ja Parso teegi poolt loodud poolikust süntaksipuust kindlat tüüpi tippude leidmine. Kogu nimekirjaga utiliitidest saab tutvuda projekti koodirepositooriumis (Lisa I) failis *error_explainer/utils.py*.

Prototüübi töövoo saab jagada kolmeks osaks, millest esimeseks on *@add_check* annotatsiooniga meetodite kokku kogumine. Meetodid paigutatakse kahte kahetasemelisse sõnastikku. Esimene sõnastik on *check* tüüpi meetodite jaoks ja teine *force_check* tüüpi meetodite jaoks.

Sõnastikud ise on üles ehitatud järgmiselt. Esimesel tasemel jagatakse meetodid annotatsioonis ette antud tasemete järgi, võtmeks saab taseme number ja väärtuseks sõnastik sellel tasemel asuvatest meetoditest. Teisel tasemel on võtmeteks meetodite nimed, millele vastavad meetodi enda objektid. Selline sügav struktuur on vajalik, et hiljem oleks meetodeid võimalik tasemete kaupa käivitada. Visuaalne näide sõnastiku ülesehitusest on toodud joonisel 3.

Teine osa töövoost on Pythoni *ast* mooduli abil ette antud faili kompileerimine, et välja selgitada, kas *check* tüüpi meetodeid käivitada. Esmalt proovitakse failist loetud koodi *ast.parse* meetodi abil kompileerida ning kui meetod tagastas vea, siis järelikult ei olnud ette antud failis kompileeruv Pythoni programmikood ja *check* tüüpi kontrollid peab käivitama.

Viimaks vastavalt sellele, kas koodi kompileerimise kontroll läbiti või mitte, käivitatakse sobivad meetodid *check* ja *force_check* sõnastikest. Meetodid käivitatakse tasemete järgi alates väikseimast. Kui kood ei kompileerunud, siis käiakse esmalt läbi *check* tüüpi meetodid seni, kuni jõutakse tasemeni, kus leitakse esimene viga. Pärast vea leidmist käivitatakse samal tasemal olnud veel käivitamata meetodid ja seejärel liigutakse edasi *force_check* meetodite juurde.

```
0: {
    'docstring_error_check': Check(level: 0, function: <function
       docstring_error_check at 0x7f9e2a63a400>),
    'quote_errors_check': Check(level: 0, function: <function
       quote_errors_check at 0x7f9e2a63a510>)
   },
1: {
    'miss_matched_bracket_check': Check(level: 1, function: <function
        miss_matched_bracket_check at 0x7f9e2a63a950>),
    'missing_brackets_check': Check(level: 1, function: <function
       missing_brackets_check at 0x7f9e2a63a840>),
    'missing_brackets_print_check': Check(level: 1, function: <
       function missing_brackets_print_check at 0x7f9e2a63aa60>)
   },
2: {
    'invalid_function_def_check': Check(level: 2, function: <function
        invalid_function_def_check at 0x7f9e2a63a730>)
   },
3: {
    'invalid_assignment_check': Check(level: 3, function: <function
       invalid_assignment_check at 0x7f9e2a63ac80>),
    'missing_colon_check': Check(level: 3, function: <function
       missing_colon_check at 0x7f9e2a63ab70>)
   }
```

Joonis 3. Check tüüpi kontrollide visuaalne representatsioon

Selles sõnastikus käiakse alates madalamast tasemest uuesti läbi meetodid, kuni järgmise vea leidmiseni või jõudes tasemeni, milles tuvastati eelnevalt *check* tüüpi viga.

Kui kood aga kompileerub, siis käivitatakse tasemete kaupa ainult *force_check* tüüpi meetodid esimese vea leidmiseni, mille korral käitutakse samamoodi nagu eelnevalt kirjeldatud *check* meetodite käivitamisel.

Kontrollidena defineeritud meetodites kasutatakse *add_message* meetodit globaalsesse nimekirja veateadete lisamiseks. Näide joonisel 4. Meetod võtab sisendiks sõne kujul unikaalse inimloetava identifikaatori, mille järgi otsitakse eelnevalt defineeritud veateadete sõnastikust vastav teade. Lisaks saab valikuliste nimeliste parameetritega (*keyword arguments*) ette anda veateadetes defineeritud muutujate väärtuseid, mis teate tekstis enne tagastust asendatakse. Kui kõik annoteeritud meetodid on käivitatud, tagastab *run_checks* meetod kasutajale nimekirja sõne kujul veateadetest.

```
@add_check(False, 1)
def missing_brackets_print_check(filename: str) -> NoReturn:
    found_errors = find_error_nodes(filename)
    for error in found_errors:
        if check_print_missing_brackets(error):
            add_message(
                "missing_brackets.print",
                 line_start=get_line_location_start(error)
            )
```

Joonis 4. Näide dekoraatori kasutamisest kontrolli lisamiseks

4.5 Prototüübi kasutamine

Teegi kasutamiseks tuleb see esmalt läbi Pythoni paketihaldus tarkvara (*pip*) endale paigaldada käsuga *pip install error-explainer*.

Pärast teegi paigaldamist tuleb see Pythoni faili importida ja seejärel välja kutsuda kontrolle käivitav meetod *run_checks* viitega failile, kus asub kood, mida kontrollida soovitakse. Näide *run_checks* meetodi kasutamisest on joonisel 5.

```
from error_explainer.check_runner import run_checks

teated = run_checks("tee/kontrollitava/failini")
```

Joonis 5. Näide kontrollide käivitamisest

Kontrolle käivitav meetod tagastab nimekirja sõne kujul veateadetest, mis vastavad ette antud failist leitud vigadele.

Arendatud teek pakub uute kontrollide lisamiseks ka dekoraator-mustrit ja mitmeid utliite, mis võivad kasuks tulla erinevate vigade otsimisel. Näide dekoraator-mustri abil kontrolli lisamisest on kujutatud joonisel 4. Olemasolevaid kontrolle on võimalik ka vajadusel nime järgi eemaldada jooksutatavate kontollide nimekirjast.

Lisaks kontrollidele saab ise lisada, eemaldada ja ümber defineerida ka veateateid kasutades joonisel 6 kujutatud meetodeid.

```
from error_explainer.messages import add_message, remove_message,
    overwrite_message

# Lisamine
create_message("code_for_the_message", "Message text with {
    dynamic_arguments}")

# Eemaldamine
remove_message("code_for_the_message")

# Defineerimine
overwrite_message("code_for_the_message", "Message text with {
    dynamic_arguments}")
```

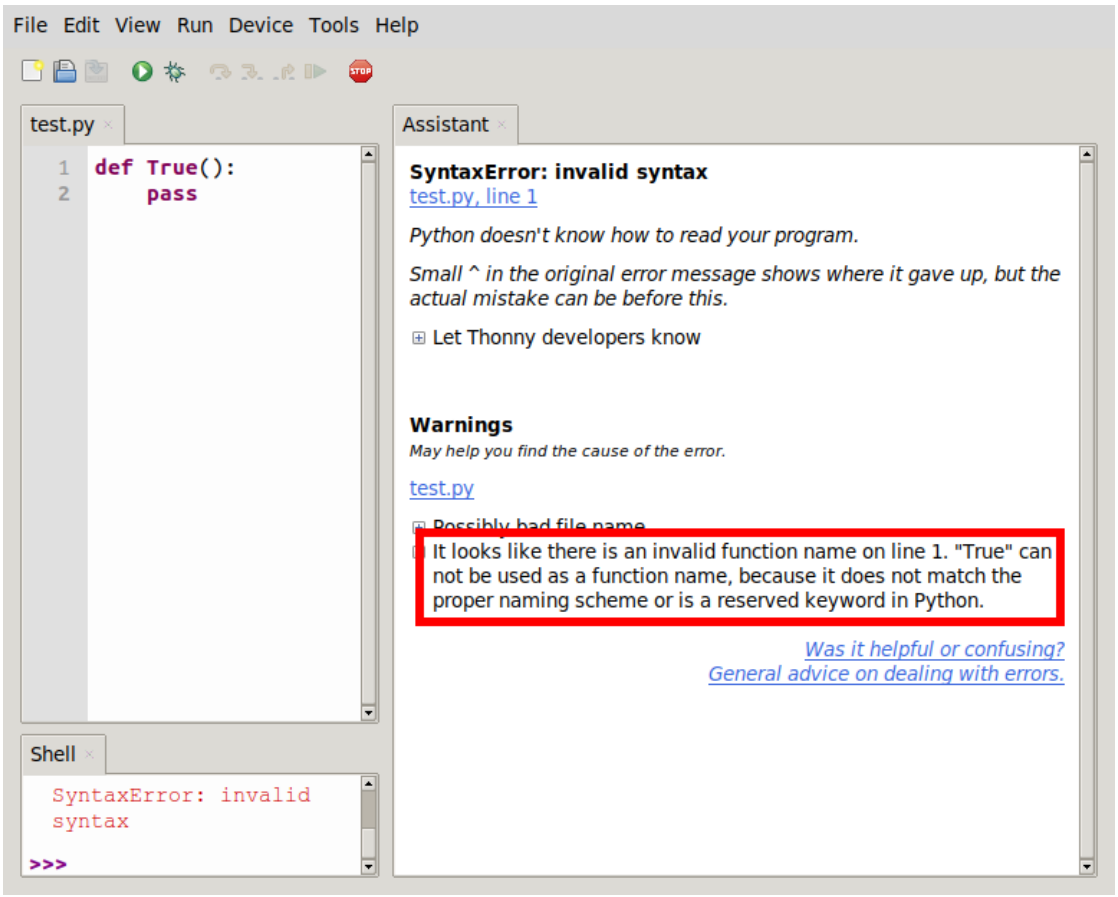
Joonis 6. Näide veateadete lisamisest eemaldamisest ja ümber defineerimisest

4.6 Thonny pistikprogramm

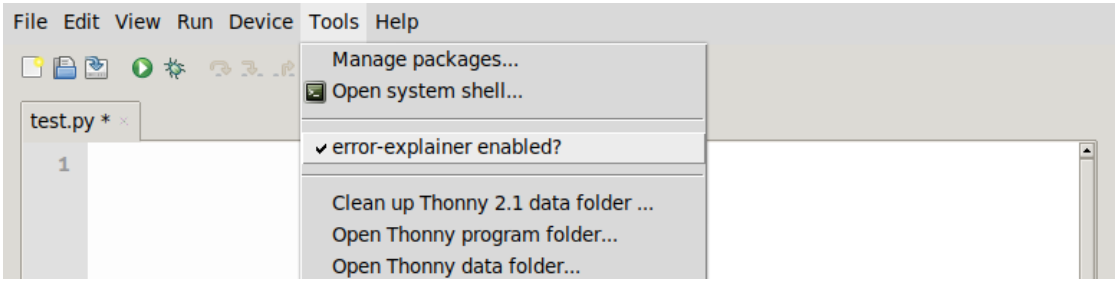
Lisaks põhiteegile valmis töö raames ka arenduskeskkond Thonny jaoks eraldi pistikprogramm, mis kasutab valminud prototüüpi, et parendatud veateateid läbi Thonny kasutajaliidese kuvada, näide joonisel 7. Kirjeldatud teek on samuti avaldatud PyPi-s nime all *thonny-error-explainer*. Thonnyle suunatud teegi arendusel on järgitud Thonny wiki lehel [19] kirjeldatud nõudeid ja tänu sellele saab pistikprogrammi mugavalt arenduskeskkonda paigaldada *Tools -> Manage plug-ins...* menüü kaudu. Pistikprogrammi lähtekood ja PyPi lehekülg on lisades III ja IV.

Pistikprogramm kasutab Thonny poolt pakutavad *ProgramAnalyzer* klassi, mida laiendades on võimalik kasutajaliideses *Assistant* aknas veateateid kuvada. Lisaks on kasutatud abimeetodit, mis võimaldab Thonny menüüdesse valikuid lisada ja selle abil on loodud nupp joonisel 8, mille kaudu on *error-explainer* teegi teateid võimaik sisse ja välja lülitada.

Kuna Thonnys esines probleem, et *Assistant* aknas kuvatud tekst lõigati reavahetuste juurest ära, siis on *thonny-error-explainer* teegis kõigis veateadetes reavahetused asendatud tühikutega. Vea kohta on avatud ka Thonny Githubis pilet [20].



Joonis 7. *error-explainer* veateade Thonny *Assistant* aknas



Joonis 8. *error-explainer* teadeadete lüliti Thonny Tools menüüs

4.7 Kontrollid

Siin jaotises on välja toodud kõik töö raames implementeeritud kontrollid. Iga kontrolliga kaasneb üks võimalik rikkumine ja näide ühest tagastatavast veateatest. Võrdluseks on toodud ka hetkel Pythonis eksisteeriv veateade. Rohkem koodinäiteid iga kontrolli poolt tuvastatavate vigade kohta on leitavad projekti koodirepositooriumis (Lisa I) üksustestidena kataloogis *test*. Kogu nimekiri realiseeritud veateadetest on samas repositooriumis failis *error_explainer/messages.py*.

docstring_error_check

Kontrollitakse, kas on üritatud kolmekordsete jutumärkide või ülakomade abil luua *docstring*, kuid on tehtud viga, kus üks või kaks lõpetavatest sümbolitest on puudu.

Näide ühest võimalikust rikkumisest:

```
"""
This is a docstring
""
```

Pythoni praegune veateade:

```
File "/path/to/file", line 4
    ""
     ^
SyntaxError: EOF while scanning triple-quoted string literal
```

***error_explainer* veateade:**

```
There seems to be missing quotes at the end of the docstring starting
    on line 1
```

quote_errors_check

Kontroll katab kaks veatüüpi, sest mõlemad neist on seotud jutumärkidega. Samuti töötab nende tuvastamise loogika hästi ühe meetodina ning seetõttu on nad kombineeritud ühte kontrolli. Vead mida tuvastatakse on järgmised:

1. sõne on alustatud, aga lõpetavat sümbolit ei tuvastata;
2. sõne on alustatud ühe sümboliga, aga lõpetatud on teisega.

Siin kontrollitakse ka kolmekordse ülakoma ja jutumärgi puhul puuduvaid sümboleid, see on ka põhjus, miks eelnev kontroll sellega ei tegele.

Näide ühest võimalikust rikkumisest:

```
a = "b
```

Pythoni praegune veateade:

```
File "/path/to/file", line 1
    a = "b
        ^
SyntaxError: EOL while scanning string literal
```

***error_explainer* veateade:**

```
There is a missing quote of type: " that should close the string
   started on line 1.
```

invalid_function_def_check

See kontroll analüüsib, kas funktsiooni definitsioonis on tehtud viga. Kaetud on järgmised juhud:

1. funktsiooni või mõne argumendi nimena kasutatakse Pythonis reserveeritud võtmesõna või ebasobivat nime;
2. funktsiooni definitsioonis puudub nimi ehk *def* võtmesõnale järgneb kohe alustav sulg;
3. funktsiooni definitsioonis puuduvad argumendid koos neid ümbritsevate sulgudega;
4. *def* võtmesõnale on üritatud väärtust omistada.

Näide ühest võimalikust rikkumisest:

```
def True():
    pass
```

Pythoni praegune veateade:

```
File "/path/to/file", line 1
    def True():
           ^
SyntaxError: invalid syntax
```

***error_explainer* veateade:**

```
It looks like there is an invalid function name on line 1.
"True" can not be used as a function name, because it does not match
   the proper naming scheme or is a reserved keyword in Python.
```

missing_brackets_check

Kontroll uurib, kas esineb probleeme sulgude tasakaalus. Kaetud on kõik kolm Pythonis kasutusel olevat sulu tüüpi (tavalised, kandilised ja loogelised). Selle kontrolli puhul esineb mõningaid probleeme alustava puuduva sulu tuvastamisel. Probleemi on põhjalikumalt kirjeldatud peatükis 5.2.

Näide ühest võimalikust rikkumisest:

```
a = [1, 2, 3
```

Pythoni praegune veateade:

```
File "/path/to/file", line 1
    a = [1, 2, 3
               ^
SyntaxError: unexpected EOF while parsing
```

***error_explainer* veateade:**

```
It looks like there are 1 missing closing square bracket(s) "]" on
   line 1
```

mismatched_bracket_check

Siin kontrollis vaadatakse, kas kasutajal on segamini läinud erinevat tüüpi sulud. Taaskord on kaetud kõik kolm Pythonis kasutusel olevat sulu tüüpi.

Näide ühest võimalikust rikkumisest:

```
a = [1, 2, 3)
```

Pythoni praegune veateade:

```
File "/path/to/file", line 1
    a = [1, 2, 3)
                ^
SyntaxError: invalid syntax
```

error_explainer **veateade:**

```
It looks like there is a mix of square and regular brackets used
   beginning on line 1 and ending on line 1.
Square brackets are used for list definitions as well as getting
   elements from a collection or string.
Regular brackets are used for tuple definitions function definitions
   and defining the order of operations in an expression.
```

missing_brackets_print_check

Jällegi on tegu sulgude vea kontrolliga. Tuvastatakse viga, kus *print* meetodit on kasutatud ilma sulgudeta, stiilis, mis on lubatud Python 2 süntaksis. Kuigi Pythoni enda veateade on antud olukorras juba hea, siis otsustati see kontroll siiski lisada. Põhjuseks oli see, et Parso poolt loodud süntaksipuus oli sellise vea puhul väga spetsiifiline tipp ja tänu sellele oli kontrolli loogika reliseerimine triviaalne.

Näide ühest võimalikust rikkumisest:

```
print 'foo'
```

Pythoni praegune veateade:

```
File "/path/to/file", line 1
    print 'foo'
              ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean
   print('foo')?
```

***error_explainer* veateade:**

```
It looks like you forgot the brackets after a print statement 1.
Print is a function and should be followed by a set of brackets like
   so: print("foo").
```

missing_colon_check

Selle kontrolliga vaadatakse, kas kasutaja on mõne komposiitlause päise järelt ära unustanud kooloni. Kontrolli poolt toetatud komposiitlaused on need, mille jaoks on Parso poolt realiseeritud eraldi süntaksipuu tipuklassid. Kogu nimekiri asub koodirepositooriumis (Lisa I) failis *error_explainer/colon_statements.py*.

Näide ühest võimalikust rikkumisest:

```
a = 1
if a > 1
    pass
```

Pythoni praegune veateade:

```
File "/path/to/file", line 2
    if a > 1
           ^
SyntaxError: invalid syntax
```

***error_explainer* veateade:**

```
It looks like you forgot to add a colon after a statement that should
    be followed by one on line 2.
if statements start a new indented code block and should be followed
   by a colon.
```

invalid_assignment_check

See on kontroll, mis katab mitmeid juhtusid, mida saab liigitada kategooriasse: vead omistuslausetes. Täpsemalt on kaetud järgnevad juhud:

1. omistatud on Pythoni võtmesõnale;
2. omistatud on funktsiooni väljakutsele;
3. omistuslause järjekord on vale (väärtus on enne võrdusmärki ja muutuja nimi pärast);
4. omistatud on konstandile nagu sõne, number või tõeväärtus.

Näide ühest võimalikust rikkumisest:

```
True = 1
```

Pythoni praegune veateade:

```
File "/path/to/file", line 1
    True = 1
    ^
SyntaxError: can't assign to keyword
```

error_explainer **veateade:**

```
Invalid assignment "True = 1" on line 1
Assignments should be in the format of variable_name,variable_name
   ,... = expression.
```

indentation_errors_check

Siin kontrollis otsitakse erinevaid taanetega seotud vigasid. Kaetud on juhud:

1. pärast komposiitlause päist pole alustatud uut taande taset;
2. taande tase on liiga kõrge ehk uus tande tase on alustatud ilma komposiitlause päiseta;
3. taande tase ei klapi ühegi varasemalt alustatud taande tasemega;
4. faili lõpus on komposiitlause päis, millele ei järgne taanet ehk komposiitlause päis on viimane rida failis.

Näide ühest võimalikust rikkumisest:

```
while True:
print("foo")
```

Pythoni praegune veateade:

```
File "/path/to/file", line 2
    print("foo")
        ^
IndentationError: expected an indented block
```

***error_explainer* veateade:**

```
There is an error in the indentation on line number 2 ("print("foo")"
    ).
No new indentation started after a statement, that should start a new
     block ("while True:").
```

coma_instead_of_period_check

Viimaseks on kontroll, mille idee on tuvastada, kas kasutaja on kümnendmurru kirjutamisel teinud vea ja kasutanud punkti asemel koma. Probleem osutus tegelikkuses keerulisemaks kui alguses tundus, sest erinevaid olukordi numbri kasutamiseks on palju. Lisaks ei pruugi see viga alati koodi süntaktiliselt ebakorrektseks muuta. Nendel põhjustel ja ajapuuduse tõttu on töö raames realiseeritud vaid juht, kus potentsiaalne viga esineb lihtsas omistuslauses.

Näide ühest võimalikust rikkumisest:

```
a = 3,4
```

Pythoni praegune veateade:

Python antud vea puhul viga ei tuvasta, kuna tegemist võib olla ka enniku omistamisega.

***error_explainer* veateade:**

```
It looks like you might have used a coma instead of a period while
    writing a fraction on line 1
```

5 Tulemused ja järeldused

Järgnev peatükk kirjeldab, kuidas loodud teeki on valideeritud ja seda, millised olid valideerimise tulemused. Lisaks võetakse kokku teadaolevad puudused tarkvaras, kirjeldatakse, mida tarkvarara loomise käigus õpiti ning lõpetuseks pakutakse välja võimalikud edasiarendusi.

5.1 Valideerimine

Tulemuste valideerimiseks kasutati Wong et al. poolt loodud andmestikku [6], kus on kokku kogutud ja sildistatud erinevad Pythonis tehtud vead. Esialgne andmestik oli üpris mürarikas ja vajas korralikku puhastust, et seda antud töö raames oleks võimalik kasutada. Valideerimiseks kasutati *error_explainer* versiooni 0.10 ja hiljem parandatud tulemusteks versiooni 0.12.

Esmalt sorteeriti veatüübi järgi andmestikust välja 100 *IndentationError* ja 200 *SyntaxError* tüüpi viga, mis vastaksid antud töö skoobile. Täpsemalt otsiti vigu, mis vastaksid järgnevatele kriteeriumitele:

1. **Koodijupi pikkus ei tohi ületada 5 rida:** Reapiirang seati, kuna algajate kood on reeglina üpris lühike. Lisaks eemaldab reapiirang ka palju selliseid koodijuppe, mis sisaldavad suurtes kogustes vigu. Samas jätab 5 rida piisavalt ruumi, et koodijupil oleks ka mingi sisu olemas.

2. **Koodijupil tohib andmestikus olla maksimaalselt 2 silti:** Siltide kogusele rakendatud piirang valiti, kuna siltidega tähistati andmestikus tihti koodis kasutusel olevaid teeke ning nende piiramine välistas suuremas osas koodijupid, kus on kasutusel edasijõudnutele suunatud teegid nagu *django*, *numpy* ja *keras*.

3. **Üks siltidest peab olema *"python-3.x"*:** See piirang rakendati, kuna teek on loodud pidades silmas Python 3.6 ja kõrgemate versioonide tuge.

Kirjeldatud piiranguid kombineerides üritati leida koodijupid, mis jäljendaksid võimalikult palju algajate poolt kirjutatud koodi.

Pärast esialgsetele tingimustele vastavate koodijuppide leidmist vaadati läbi alles jäänud näited. Neist eemaldati sellised, mis ei olnud üldse Pythoni kood ning need, milles esines vigu, mida teek ei toetanud. Pärast käsitsi kontrollimist jäi algsest 300 koodijupist alles 128. Nende 128 peal rakendati lihtsat skripti, mis salvestas iga koodijupi eraldi faili koos teegi poolt pakutud veateatega. Esialgsed tulemused on toodud tabelis 1.

Tulemus	Kogus
Õige	71
Tuvastati viga taandes, mille asukoht eksis ühe rea võrra	40
Õige puuduva sõne sümboli viga, aga asukohaviide real vale	9
Tuvastati vale viga	3
Viga oli aga seda ei tuvastatud	3
Tuvastati vigane definitsiooni vorm, tegelik viga oli puuduv koolon	1
Viga sulgude kasutuses, aga sulu tüüp vale	1

Tabel 1. Esialgsed valideerimise tulemused

Esmaseid valideerimistulemusi uurides selgus, et suur osa (49) eksimusi tuli kahest väga kergesti parandatavast veast teegi realisatsioonis. Täpsemalt vead tabelis 1 ridadel 2 ja 3. Esmalt eksimus, kus teek tuvastas vea taandes, kuid eksis vea asukohaga 1 rea võrra. See eksimus esines ainult ühte tüüpi taande vea puhul (viga, kus komposiitlausele ei järgnenud taandatud rida). Teiseks eksimus, kus teek tuvastas õigesti vea puuduvas sõne alustavas või lõpetavas sümbolis, kuid eksis vigase sümboli asukohaga real.

Esimese eksimuse parandamiseks oli vaja lihtsalt abimeetodi loogikas reanumbrist 1 lahutada. Teise eksimuse parandamiseks otsustati eemaldada veatedetes viide rea sisesele asukohale, sest tegelikkuses on väga raske otsustada, milline ülakoma või jutumärk on täpselt see, millel puudub paariline. Lisaks on asukoht real üsna ebaoluline informatsioon, kuna täpselt viidatud asukoha leidmine on üsna ebatõhus ja aeganõudev tegevus ning õpetaks algajatele halbu harjumusi. Pärast kirjeldatud pisikesi korrektuure nägid tabelis 2 välja toodud tulemused välja hoopis paremad.

Tulemus	Kogus
Õige	120
Tuvastati vale viga	3
Viga oli aga seda ei tuvastatud	3
Tuvastati vigane definitsiooni vorm, tegelik viga oli puuduv koolon	1
Viga sulgude kasutuses, aga sulu tüüp vale	1

Tabel 2. Parandustega valideerimise tulemused

Lisaks valideerimisele, kus kasutati andmeid, mida autor varem näinud ei olnud, on teegi koodibaasis olemas ka üksustestid iga kontrolli kohta. Üksustestid on aga terve rakenduse arenduse käigus autorile teada olnud ja nende abil on regulaarselt koodi katsetatud. Üksustestides kasutatud koodinäited on pärit varasemalt mainitud teegi *friendly_tracebak* [16] repositooriumist või autori enda poolt välja mõeldud.

5.2 Teadaolevad puudused

Valideerimise käigus selgusid mõningad puudused teegi funktsionaalsuses, mille parandamine ei olnud nii triviaalne, kui valideerimise jaotises kirjeldatu. Esimeseks selliseks puudujäägiks on see, et funktsiooni definitsiooni puhul ei tuvastata puuduva koolooni viga *missing_colon_check*, vaid hoopis *invalid_function_def_check* kontrolli poolt. See puudujääk ei ole tegelikkuses kuigi kriitiline, kuna tagastatud veateade on *invalid_function_def_check* kontrolli puhul ka tehniliselt korrektne, aga mitte nii täpne, kui oleks *missing_colon_check* kontrolli puhul.

Teine puudujääk olemasolevates kontrollides on see, et puuduva alustava sulu tuvastamine on ebatõhus. Nimelt jätab Parso vigase tipu tagastamisel välja info üksiku sulu kohta. Näiteks järgneva koodijupi puhul:

```
a = 1, 2, 3]
```

loob Parso tipu:

```
<PythonErrorNode: a = 1, 2, 3@1,0>
```

Tipus on täielikult välja jäetud info sulgude kasutuse kohta koodis, sellest tulenevalt on keeruline vea kohta mõistlikku pakkumist teha.

Lisaks on veel mõningaid äärejuhtusid, kus tuvastatakse vale viga ning olukordi, kus teek üldse viga tuvastada ei suuda. Autorile teadaolevalt on kõik need eksimused põhjustatud ühe disainiotsuse tõttu. Nimelt toetuti tarkvara loomisel liialt Parso teegile. Teek oli väga suureks abiks suurema osa kontrollide loomisel. Küll aga eksis teatud olukordades piisavalt, et korrektse tulemuseni jõudmiseks oleks pidanud täiesti eraldi lahenduse realiseerima.

5.3 Tarkvara loomise käigus omandatud teadmised

Ainuüksi süntaksivigade puhul on vigade tuvastamise ja veateadete parandamise näol tegemist väga mahuka valdkonnaga. Süntaksis on palju erinevaid nüansse, mida kõike on tagantjärgi koodianalüüsi kaudu praktiliselt võimatu jälgida ja kontrolle luues silmas pidada. Lisaks on probleemi lahendust toetavaid vahendeid töö realiseerimise hetkel äärmiselt vähe ja need, mis eksisteerivad, ei ole alati täielikult töökindlad. Kirjeldatud asjaolusid arvestades võiks öelda, et vigade tuvastamine, kasutades Pythonit, on üpris kohmakas ja ebatõhus. Tõhusama lahenduse välja töötamiseks tuleks arvatavasti probleemile läheneda läbi spetsiaalse parseri väljatöötamise, kuna nii on probleemi võimalik lahendada palju süstemaatilisemalt ja elegantsemalt.

Lisaks selgus, et algajasõbralikumate veateadete loomine võib olla ka üpris keerukas. Algajasõbraliku tarkvara loojad on kogenenumad arendajad, kelle jaoks on paljud algajate valukohad suure kogemustebaasi tõttu väga võõraks jäänud. Seetõttu on neil ka raske tabada täpselt seda, mis teeb ühe programmeerimiskeele õppimise keeruliseks ja veateated ebaselgeks. Isegi, kui on olemas varasemad uurimused, kus analüüsitakse,

miks teatud veateated on kehvad, siis informatsioon selle kohta, kuidas neid algajate jaoks parendada, on puudulik ja vajab veel täiendust.

5.4 Võimalikud edasiarendused

Tarkvara loomisel on pandud suur rõhk paindlikkusele ja edasiarendamise toetamisele. Tänu sellele oleks käesoleva töö raames loodud teek hea põhi, mille pealt uusi kontrolle edasi arendada. Nii laieneks kaetud vigade arv oluliselt ja järgnevad panused probleemi lahendamisele saaksid ära kasutada juba olemasolevat funktsionaalsust. Loodud teek sobiks hästi keskseks baasiks, kuhu tulevikus sarnaseid arendusi kokku koguda. Nii ei ole järgnevad panused fragmenteeritud ja lõpuks on ühine kasutegur oluliselt suurem.

Üks võimalikest edasiarendustest oleks selles töös loodud veateadete tõhususe valideerimine, mis antud töö skoobist välja jäi. Võimalus oleks võtta loodud tarkvara läbi arenduskeskkonna Thonny kasutusele mõnes algajatele suunatud programmeerimise kursuses. Kursust läbi viies koguda infot ja analüüsida, kui tõhusad uued teated on ning kuidas neid täiustada. Võimalik edasiarendus oleks ka olemasolevate kontrollide parendamine ja teadaolevate puuduste likvideerimine.

Töö kaigus tekkisid ka mõningad ideed, mis hetkel pole realiseeritud, kuid mida tasuks katsetada. Põhilise funktsionaalsuse poolelt tekkis kaks ideed. Esiteks oleks algajatele suunatud teegi juures hea, kui oleks toetatud veateadete tõlkimine kasutaja emakeelde – hetkel sellist funktsionaalsust pole. Ning teiseks idee, mida põgusalt katsetati, aga mis osutus hetkel liiga mahukaks. Nimelt võiks teha katsetusi tuvastatud vea põhjal automaatselt koodi parandamises ning selle abil valideerida, kas teegi poolt tehtud pakkumine oli õige või mitte. Selle töö käigus katsetati ideed puuduva kooloni sisestamisel, kuid lõpuks otsustati, et vajalik arendus ei paku hetkel piisavalt lisaväärtust, arvestades selle mahukust ja veaohtlikkust.

Lisaks kogunes valideerimisandmeid läbi töötates kaks ideed uute kontrollide jaoks, mille realiseerimiseni ei jõutud. Esiteks kontroll, mis tuvastab viga, kus kasutaja on sõnesid ja muutujaid omavahel kokku liites ära unustanud plussmärgi. Teiseks kontroll, mis leiab üles teistes programmeerimiskeeltes kasutusel olevate kommenteerimisstiilide kasutuse.

6 Kokkuvõte

Töö raames valmis tarkvaraprototüüp, mis pakub paindlikku raamistikku erinevate lähtekoodi kontrollivate meetodite implementeerimiseks ja nende põhjal teadete tagastamiseks. Valminud raamistik on eelkõige suunatud Pythoni-põhiste tööriistade loojatele.

Valminud teegiga kaasnevad ka mitmed Pythonis algajate poolt tehtud süntaksivigasid tuvastavad kontrollid ja neile vastavad veateated. Koostatud teated proovivad anda Pythoni kompilaatorist algajasõbralikumat tagasisidet. Selleks on loodud veateated võrdlemisi lühikesed ja sisaldavad minimaalselt žargooni.

Kuna rakendusega kaasnevad kontrollid on suunatud algajatele, siis on lisaks realiseeritud ka pistikprogramm arenduskeskkonna Thonny jaoks, mis võimaldab kontrolle automaatselt käivitada ja nende abil läbi kasutajaliidese programmeerijale tehtud süntaksivigade kohta tegasisidet anda. Nii teek kui ka pistikprogramm on realiseeritud Pythonis ja kaasnevad kontrollid toetuvad suuresti pooliku süntaksipuu loomist toetavale, vigadest taastuvale parserile Parso.

Lisaks prototüübi realisatsioonile on töö raames eksperimentaalselt hinnatud, kui tõhus vigadest taastuva parseri-põhine lähenemine veateadete parendamiseks võiks olla. Välja on toodud ka implementatsiooni käigus avastatud puudujäägid.

Hetkel on implementeeritud varasema kirjanduse põhjal valitud süntaksivigade kontrollid, kuid teegi paindlikkuse ja juba realiseeritud utiliitide tõttu oleks see sobilik keskus, kuhu edasiste tööde raames kontrolle juurde luua. Nii kasvaks tulevikus oluliselt tarkvara kasulikkus ja kaetud vigade kogus.

Viidatud kirjandus

[1] Raigo Kodasmaa. "Programmeerimiskeele Python veateated programmeerimise algõppes". Magistritöö. 2017. URL: https://dspace.ut.ee/handle/10062/65819.

[2] David Pritchard. "Frequency distribution of error messages". *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools* (2015), l. 1–8. DOI: 10.1145/2846680.2846681.

[3] Anne K. Kelley. "A system for classifying and clarifying Python syntax errors for educational purposes". Magistritöö. Massachusetts Institute of Technology, 2018. URL: https://dspace.mit.edu/handle/1721.1/119750.

[4] Titus Barik et al. "Do developers read compiler error messages?" *Proceedings of the 39th International Conference on Software Engineering* (2017), l. 575–585. DOI: 10.1109/ICSE.2017.59.

[5] James Prather et al. "On Novices' Interaction with Compiler Error Messages: A Human Factors Approach". *Proceedings of the 2017 ACM Conference on International Computing Education Research* (2017), l. 74–82. DOI: 10.1145/3105726.3106169.

[6] Alexander William Wong et al. "Syntax and Stack Overflow: A Methodology for Extracting a Corpus of Syntax Errors and Fixes". *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (september 2019), l. 318–322. ISSN: 2576-3148. DOI: 10.1109/ICSME.2019.00048.

[7] Mark Lutz. *Learning Python*. 1. väljaanne. USA: O'Reilly & Associates, Inc., 1999.

[8] Aivar Annamaa. "Introducing Thonny, a Python IDE for learning programming". *Proceedings of the 15th Koli Calling Conference on Computing Education Research* (2015), l. 117–121. DOI: 10.1145/2828959.2828969.

[9] John Magee. *Python Debugging (fixing problems) A guide for beginner programmers.* URL: https://www.cs.bu.edu/courses/cs108/guides/debug.html (külastas 02. 12. 2019).

[10] Tobias Kohn. "The Error Behind The Message: Finding the Cause of Error Messages in Python". *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (2019), l. 524–530. DOI: 10.1145/3287324.3287381.

[11] Marie-Hélène Nienaltowski, Michela Pedroni ja Bertrand Meyer. "Compiler error messages: what can help novices?" *Proceedings. of. the. 39th. SIGCSE technical. symposium. on. Computer. science. education.* 40.1 (märts 2008), l. 168–172. ISSN: 0097-8418. DOI: 10.1145/1352135.1352192.

[12] Brett A. Becker, Kyle Goslin ja Graham Glanville. “The Effects of Enhanced Compiler Error Messages on a Syntax Error Debugging Test”. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (2018), l. 640–645. DOI: 10.1145/3159450.3159461.

[13] Brett A. Becker. “An Exploration Of The Effects Of Enhanced Compiler Error Messages For Computer Programming Novices”. *ARROW@TU Dublin* (2015). URL: https://arrow.tudublin.ie/ltcdis/35.

[14] Tobias Kohn. *Arenduskeskkonna TygerJython koduleht*. 2014. URL: http://jython.tobiaskohn.ch/ (külastas 22. 03. 2020).

[15] Tobias Kohn. *TigherJythonis kasutusel oleva parseri Github-i lehekülg*. URL: https://github.com/Tobias-Kohn/TigerPython-Parser (külastas 22. 03. 2020).

[16] André Roberge. *Friendly Traceback teegi dokumentatsioon*. URL: https://aroberge.github.io/friendly-traceback-docs/docs/html/ (külastas 22. 03. 2020).

[17] David Halter. *parso*. URL: https://github.com/davidhalter/parso (külastas 27. 04. 2020).

[18] *PEP 318 – Decorators for Functions and Methods*. Mai 2020. URL: https://www.python.org/dev/peps/pep-0318 (külastas 04. 05. 2020).

[19] Aivar Annamaa. *Thonny pistikprogrammide wiki lehekülg*. URL: https://github.com/thonny/thonny/wiki/Plugins (külastas 27. 04. 2020).

[20] Kaarel Loide. *Thonnys avastatud vea pilet*. [Online; accessed 4. May 2020]. Mai 2020. URL: https://github.com/thonny/thonny/issues/1186 (külastas 04. 05. 2020).

Lisad

I. Prototüübi github repositoorium

Prototüübi koodirepositoorium, koos töö raames valminud veateadete ja üksustestidega asub järgneval veebiaadressil. Koodirepositooriumis asuvas *README.md* failis on ka juhend teegi kasutamiseks.

`https://github.com/K44rel/error-explainer`

II. Prototüübi PyPi veebileht

Pythoni paketirepositooriumis PyPi asuv veebileht loodud prototüübi jaoks.

`https://pypi.org/project/error-explainer`

III. Thonny pistikprogrammi github repositoorium

Koodirepositoorium töö käigus loodud teegi põhise Thonny pistikprogrammi jaoks.

`https://github.com/K44rel/thonny-error-explainer`

IV. Thonny pistikprogrammi PyPi veebileht

Pythoni paketirepositooriumis PyPi asuv veebileht Thonny jaoks loodud pistikprogrammile.

`https://pypi.org/project/thonny-error-explainer`

V. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Kaarel Loide**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose

   Pythoni süntaksivigade analüüs algajasõbralikumate veateadete kuvamiseks,

   mille juhendajad on Aivar Annamaa ja Vesal Vojdani,

   reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, 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.

Kaarel Loide
08.05.2020