communicated by:
Lehr- und Forschungsgebiet Informatik 9

Prof Dr. Ulrik Schroeder

The present work was submitted to Learning Technologies Research Group
Diese Arbeit wurde vorgelegt am Lehr- und Forschungsgebiet Informatik 9

Contextual Automatic Code Repair for Python Programming Novices

Kontextbasierte Automatische Codereperatur für Python-Anfänger

Master-Thesis
Masterarbeit

presented by / von

Steffes, Leonard
394539

Prof. Dr. Ulrik Schroeder

Prof. Dr. Jürgen Giesl

Aachen, August 5, 2024

Contents

List of Figures

List of Tables

List of Listings

Glossary

AST	Abstract Syntax Tree	66
API	Application Programming Interface	75
CRUD	Create, Read, Update and Delete	20
CSV	Comma Seperated Values	51
JSON	JavaScript Object Notation	24
LLM	Large Language Model	24
LMS	Learning Management System	50
MOOC	Massive Open Online Course	14

Abstract

Correcting submissions to programming exercises by hand is time consuming and thus expensive. The increase in enrolments in computer science subjects and the expansion of programming courses in other degree programmes are exacerbating this problem. Thus, alternatives to manually reviewing every submission, checking its correctness and providing feedback to the students is needed. Current tools either focus on checking the correctness of a submission or provide feedback for buggy submissions. However, both is needed as a simple correct or not correct message is usually not sufficient for students. Also, feedback changing the code should only be created if the submission indeed contains errors, or it must be marked as optional feedback, e.g., improving the code style.

To solve this issue, we develop Pyrat, a Python repair tool that combines the validation of correctness of submissions and, if necessary, a program repair algorithm to generate feedback. In case of correct code, just a message that the submission is correct is returned to the student, optionally extended with coding guideline violation hints. On the other hand, if the code is not correct, test results are returned together with feedback on how to solve the error. The correctness validation is perfomed on the base of unit tests, extended with supporting libraries for static code analysis. For the repair part, a static algorithm is developed that computes the edit distance of the Abstract Syntax Tree of the buggy submission against all available correct submissions for the task. Based on the tree edit operations with the minimum cost, feedback is generated that advises the student on how to fix the error in their submission. A Large Language Model (LLM) is used to support teachers in the creation of unit tests and reference solutions for the repair.

To analyse the quality of the feedback, we perform a study analysing the feedback generated by Pyrat on buggy student submissions. Evaluating the results of the study shows that Pyrat performs well on submissions with small errors if a correct submission is available that follows a similar approach. However, the feedback is too complex if no submission is known that follows the student's approach. Thus, further research is necessary on how complex feedback can be cut down into messages that are easy to understand, e.g., by not returning the complete feedback at once but providing just a first hint instead.

Chapter 1 Introduction

1.1 Motivation

In programming education, exercises are often used to improve the learning process compared to a lecture without any training. By correcting the submitted exercises teachers can give students feedback on how well they did. However, the increase of students in introductory programming lectures makes it hard for teachers to manually provide feedback for all submissions.

To mitigate this problem, automated assessment and repair tools can be used to aid the teachers in providing feedback to the students. The capabilities' range of such tools is very high. Simple code assessment tools like Web-CAT [7] or nbgrader [16] check the correctness of exercise tasks by predefined rules. Thus, grading is possible for the teachers, e.g., to encourage students to work on the exercises by making them mandatory for admission on an exam or by improving the exam's grade. However, feedback usually consists of a simple "passed" or "not passed", sometimes extended with the results of a unit test. The problem with this is that such simple feedback is no good aid for novices [20]. One reason is the complexity of the unit tests' output.

While code assessment tools are useful to grade submissions, they fail on providing meaningful feedback to the students. As an alternative, code repair tools try to find and fix the error in a buggy submission for an exercise task. Using the results from this repair process, it is either possible to return a working code to the student or to generate feedback messages that describe the error in the program in an understandable way [14, 10]. Some of those tools, e.g., CLARA or Refactory, execute the buggy submission to generate the necessary repair. Thus, they require the code to be syntactically correct, i.e., only semantic errors can be fixed with these tools.

Another option are code repair tools based on Large Language Models (LLMs). As they do not execute the code but use the capabilities of a pre-trained LLM instead, they are also capable of fixing syntax errors. However, there is no guarantee that the feedback generated by an LLM is useful or even correct. Another drawback is the availability of LLMs. While access to commercial LLMs, e.g., OpenAI's ChatGPT[1], is often possible using an Application Programming Interface (API), this makes teaching at university level dependent on the pricing of commercial companies.

Instead of generating feedback for semantical errors, tools like PythonTA focus on providing hints regarding code violation errors [23]. Thus, students can identify logical errors and prevent semantical errors from occurring themselves.

[1] ChatGPT, `https://chatgpt.com/`, last accessed 07/24//2024

All of the tools presented in the Related Work chapter of this thesis (chapter 3) focus on one of these tasks, i.e., either they use a unit test base approach to determine if a submission is correct or not or they offer code repair to generate feedback for a buggy submission. However, just checking if the solution is correct or not does not serve as feedback; and generating feedback using a code repair algorithm requires knowledge whether the submission needs repair at all. Thus, teachers need to somehow manually combine different tools to get feedback only for those tasks that require it. To ease this task, we present Pyrat, a Python repair tool combining code assessment and a static code repair algorithm.

1.2 Research Questions

The goal of this Master's thesis is to develop a tool that can be used to automatically repair buggy code submissions. To structure the work, we focus on the research questions below.

[RQ1] Which Python features need to be supported by a code repair tool? Not all Python features are supported by current tools like CLARA or Refactory. As an example, both tools cannot work with raised exceptions [10, 14]. To build a code repair tool that fits the need of Python lectures, we discuss which features of the Python language are required for novice learners.

[RQ2] How can a code repair tool be used for new exercises? In current code evaluation and code repair tools like nbgrader, CLARA or Refactory, unit tests and reference solutions for the tasks are required to evaluate and fix the student's submissions [16, 10, 14]. As providing unit tests and a larger amount of reference solutions is time-consuming, we will discuss options to reduce this effort.

[RQ3] How can a code repair tool be integrated into existing courses and technical setups? If a Python course already exists, it might also already use an exercise system that teachers and students are used to. To support teachers and not enforce the usage of a specific exercise system, we discuss the options of integrating the developed tool into existing environments like JupyterLab [32].
To answer these research questions, we develop Pyrat, a code repair tool that integrates the assessment of submissions, feedback generation, LLM-based unit tests and reference solution creation as well as code clustering for correct submissions.

Chapter 2 Preliminaries

2.1 Abstract Syntax Trees

While programming in Python is done using a textual language, the program code is not directly interpreted. Instead, an Abstract Syntax Tree (AST) is generated first that can then be used to execute the program.
In Python, it is possible to programmatically compute the AST of any code received as an input using the built-in *ast* module[1] used by the compiler as well [33]. The computation of ASTs using the built-in module is static, i.e., the code that is to be parsed is not executed during the computation of the AST [33]. This reduces security issues as potentially malicious code is not executed and thus cannot cause any harm. Additionally, it is also possible to generate ASTs from codes that cannot be fully executed for semantic reasons, e.g., because they mistakenly implement an infinity loop. However, it is not possible to create AST from codes with syntax errors.

```
[i ** 2 for i in range(10)]
```

Listing 2.1: Small Python code as ast example

```
Module(
  body=[
    Expr(
      value=ListComp(
        elt=BinOp(
          left=Name(id='i', ctx=Load()),
          op=Pow(),
          right=Constant(value=2)),
        generators=[
          comprehension(
            target=Name(id='i', ctx=Store()),
            iter=Call(
              func=Name(id='range', ctx=Load()),
              args=[
                Constant(value=10)],
              keywords=[]),
            ifs=[],
            is_async=0)]))],
type_ignores=[])
```

Listing 2.2: AST dump of code in listing 2.1

[1] ast - Abstract Syntax Trees - Python 3.12.4 documentation, https://docs.python.org/3/library/ast.html, last accesses 07/23/2024

As an example of ASTs in Python, see the demonstration in listings 2.1 and 2.2 calculating the squares of all numbers from zero to five. Every item in the AST in listing 2.2 is an object of an according Python class, e.g., Module for the overall module or Pow for the binary power operator [33]. Thus, every property of every element in the tree, e.g., the value of the constant 2, is accessible.

2.2 APTED

The All Path Tree Edit Distance (APTED) algorithm is an efficient algorithm to calculate the tree edit distance of arbitrary trees [29] with an available Python implementation [30]. The tree edit distance is a measure for the similarity of two trees, "which is defined as the minimum-cost sequence of node edit operations that transform one tree into another" [29]. While the idea behind the implementation and the mathematical proof is not of interest for this thesis and can be found in the paper of Pawlik and Augsten, we focus on the required input and the output generated by the Python implementation [29, 30].

In general, the APTED algorithm takes two trees A and B as input and computes the update cost between both trees and a list of operations that transforms tree A into tree B [29]. When using the Python package for APTED without any configuration, trees nodes are expected to have a string label and an ordered list of children. A tree is then represented by the root node. To calculate the edit distance between A and B, the algorithm knows four different operations with a cost metric that can be performed on the tree [29, 30].

The insert operation creates a new labeled node in the tree with an arbitrary number of children. The cost is hereby defined as the length of the label, i.e., inserting a node with the label *abc* has a cost of 3. The remove operation, on the other hand, removes a labeled node from the tree [30]. However, children of the node are not necessarily removed as well. The cost is again defined as the length of the label [30].

The third operation, the update operation, changes the label of a node. This reduces the total distance as updating a label is cheaper than removing the old label and inserting a new one. In the base implementation of APTED, Levenshtein distance is used to calculate the cost of updating a label [30]. The Levenshtein distance is a string metric to calculate the cost of changing a string into another one. It counts how many single-character edits, i.e., insertions, removals and updates (as for the tree edit distance) are required to change the string.

The last operation is a match operation, it is used if the nodes in tree A and B have the same label.

Figures 2.1 and 2.2 show two example trees for which APTED can calculate the tree edit distance. While the node *date* has to be removed from tree A, a new node *honeydew* needs to be inserted as the second child of *fig*. Due to the length of the label, this results in an insertion cost of 4 and an update cost of 8, summing up to 12. The node labels *cherrie* and *cherry* have a Levenshtein distance of 2, as the *i* has to be replaced by a *y* and the *e* has to be removed. Compared to a cost of 13 for removing and re-inserting this node, this is an improvement resulting in an overall tree edit distance of 14 between tree A and B.

While this basic configuration is suitable for general calculation of tree edit distance, it is not sufficient for calculating the edit distance and operations between Python ASTs as the cost metric is not sufficient. This is because the labels of the nodes in the AST are not chosen in a way that Levenshtein distance can be used to measure the cost of inserting, removing or updating them. However, it is possible to apply a customized configuration that changes the cost metric. In detail, a configuration class with four methods is defined. While the methods `insert`, `remove` and `rename` calculate the cost of performing the specified operation on one or two nodes, respectively, the `children` method returns a list of all children of the given node [30]. Thus, we can adopt the APTED algorithm to match the needs of Pyrat's code repair algorithm.

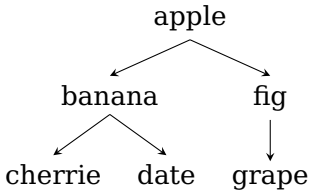


Figure 2.1: Example tree A

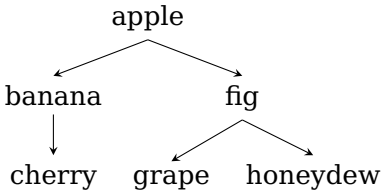


Figure 2.2: Example tree B

Chapter 3 Related Work

In the recent years, a lot of research in the domain of automated code repair and feedback generation has been published. In this chapter, we introduce existing literature and tools to better narrow down the scope for Pyrat, the PYthon and RepAir Tool developed in this thesis.

3.1 Code Assessment Tools

The most basic tools providing feedback on coding exercises for novices are code assessment tools that evaluate code submissions based on predefined rules, e.g., unit tests. While the earliest code assessment tools where developed in the 1960's [12], there are now multiple publications every year presenting and describing new code assessment tools as presented by Souza et al. [44]. 70 % of the analyzed tools work automatically, i.e., the assessment criteria is defined before the student's submissions are assessed and no further interaction by the teacher is required [44]. Of those tools, 62 % follow a student-centered approach, i.e., the validation of the submission can be initiated by the students whenever they like [44]. However, the code assessment tools listed by Souza et al. are not developed for Python [44].

A flexible code assessment and auto-grading tool supporting Python is Web-CAT [7]. Teachers can create the code assignments they want to hand out to their student's in the Web-CAT system. Students can then use a plugin in their programming environment to submit their solution to Web-CAT for grading, where feedback is generated [24]. Besides applying teacher-defined tests on the code, Web-CAT offers the option to use student-written test cases and calculate the grade on the coverage of these tests [48].

Another code assessment tool is nbgrader, a plugin for JupyterLab [16, 32]. JupyterLab is a programming environment that can be accessed by a web browser and does not need any setup on the client machine [32]. Thus, using JupyterLab in introductory programming courses has the advantage of reducing the setup hurdles for students. The nbgrader project now makes it possible to grade coding exercises directly in the notebook files, i.e., students can work online on an exercise sheet implemented as a Python notebook file and later submit this file for grading. To provide this functionality, nbgrader extends a Python notebook file by the required metadata. More precisely, code cells can be marked as automated correction cells by the teacher when creating the notebook file for the students. Code inside these automated correction cells is then considered as a unit test and removed from the version of the notebooks that is created for the students. When a notebook file is grated, all cells in the notebook are executed, including the students' submission

cells as well as the correction cells. If and only if the code in a correction cell can be executed without errors, the points assigned to this cell are granted to the student [16]. However, teachers are in charge of the collection of the notebook files from the students, performing the repair, and the returning of the results to the students. Using the classification of Souza et al., nbgrader uses an instructor-centered approach performing an automatic assessment [44].

3.2 Repair Tools

The tools presented in section 3.1 only assess the submitted code, i.e., they determine if the code is correct or not. If feedback is generated and returned to the student at all, it consists of a grade and possibly the unit test results, but not more. However, this is not necessarily useful for students [20]. In contrast to this, the repair tools presented in this section aim to generate feedback on buggy submissions that helps the students to find the error in the submission and fix it.

CLARA In their paper, Gulwani et al. present CLARA, designed to provide feedback on necessary correction steps to match a working solution for syntactically correct code [10]. Figure 3.1 shows the overall strategy of the repair algorithm. The key idea of CLARA is "to use the existing correct student solutions to repair the incorrect attempts" [10]. For an assignment, CLARA clusters all already known correct submissions for the task based on "dynamic equivalence" [10]. To be dynamically equivalent, two codes need to have the same looping structure, i.e., both need to have the same types of loops (for/while) in the same order. Also, the variables of both codes need to form a "total bijective relation" [10], meaning that when the code is executed on the same input, related variables have to take the same values in the same order [10]. This clustering reduces the number of reference solutions a buggy code needs to be compared against, as only one reference solution per equivalence set is used.

In the repair part, CLARA then compares a buggy code against every cluster of reference solutions. The repair is then created by choosing the reference solutions with the smallest syntactic distance [10].

This approach has two drawbacks as the code submitted by the student must be executed in the repair process. First, this must be done in a secure environment to prevent malicious code (whether intentional or not) from compromising the executing system. A prominent example for unintentionally harmful code is the usage of infinity loops, working like a denial-of-service attack on the executing system if not handled correctly. However, even if handled correctly, the existence of infinity loops disturbs the repair process. While it is possible to prevent problems for the executing system caused by infinity loops, e.g., by using time limits for the repair of a single buggy submission, code repair cannot deliver feedback on how to solve the problem if it is killed due to a timeout.

Second, the environment needs to be correctly set up to meet the requirements of a task. As an example, consider a task where student’s need to read data from a file and transform it based on given rules. In the repair process, it is now necessary to provide this environment for every repair. Otherwise, we could cause new errors in the code, e.g., because a file that should exist under a certain location does not exist.

However, those would not be errors that have to be repaired as they are not caused by the student submitting the exercise.

Additionally, CLARA uses an own parser for the Python codes [10]. However, this compiler does not support every feature of the current Python versions, e.g., object orientation or the match statement is not supported. Also, the number of available modules is restricted, e.g., submissions that need repair cannot work with tools like pandas[1].

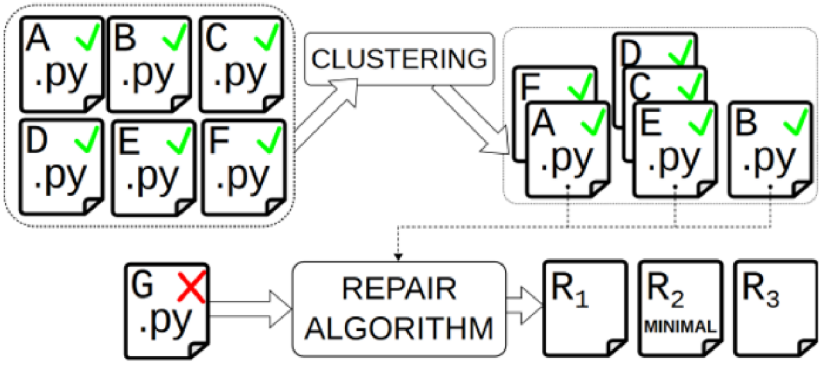


Figure 3.1: The overall procedure of CLARA [10]

Refactory To avoid the need of a large amount of correct solutions for clustering, Hu et al. developed Refactory [14]. The core idea of Refactory is to take a single expert-written sample solution and generate semantically equivalent mutations of the control flow following certain refactoring rules, e.g., transforming existing conditional statements and loop guards. As figure 3.2 shows, the first step of the correction is the structural alignment of the buggy program with one of the refactored versions. If a version of the expert solution can be refactored in a way that its structure is equivalent to the one of the buggy program, it can be repaired on this base. Otherwise, the structure of the buggy program itself needs to be mutated to match a working solution. Unlike the refactoring process of the expert solution, this does not necessarily guarantee preserving semantic equivalence [14]. Refactory is then able to suggest code repairs needed to map the chosen refactored expert solution for a buggy program.

As code repair in Refactory again relies on executing the buggy code, the same problems that are mentioned for CLARA exist for Refactory as well. Additionally, the refactoring rules are not necessarily sufficient to generate enough different reference solutions for an efficient repair. However, it is possible to include correct student submissions in the repair process, just like it is for CLARA. As Refactory misses a clustering approach, teachers need to make sure that the redundancy in the set of reference solutions does not get too high as the performance of Refactory decreases with an increase of available reference solutions [14].

[1] pandas - Python Data Analysis Library, `https://pandas.pydata.org/`, last accesses 07/23/2024

Also, Refactory delivered non deterministic results in our evaluation experiments. When repairing the same buggy submission multiple times with the exact same set of reference solutions available, Refactory was able to generate a repair in only some of the executions.

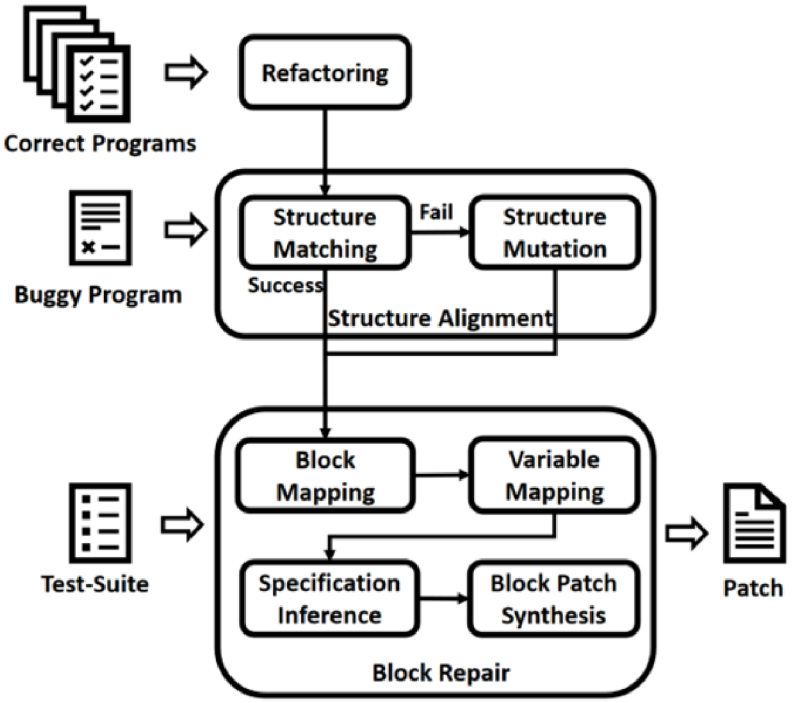


Figure 3.2: The overall procedure of Refactory [14]

RNN Based Syntax Repair A major drawback of those approaches is that they require even the wrong submissions to be syntactically correct [10, 14]. Bhatia and Singh use a Recurrent Neural Network (RNN) trained with a set of syntactically correct solutions to correct syntax and indentation errors in submissions. Trained with the syntactically correct solutions, the neural network learns a token sequence specifically for a given task. Once the compiler finds an error in one of the solutions, the tokens up to the error are fed into the model to retrieve a sequence of likely tokens that might fix the error. Each token is then applied and the result is checked for syntactic correctness [2].

MMAPR To combine the approaches of syntax and semantic code repair, Zhang et al. invented the multi-modal code repair system (MMAPR) [53]. As figure 3.3 shows, MMAPR is separated into two phases: a syntax and a semantic phase, both working with a Large Language Model Trained on Code (LLMC). In the syntax phase, the potentially buggy program is fed into a syntax oracle determining if the program contains a syntax error, and if so, the location of the (first) error. If the oracle returns a syntax error, the chunk of the error is isolated and fed into the LLMC to receive repair suggestions, once with and once without the syntax error message received from the oracle. Once all syntax errors are solved by the algorithm, the repair candidates can

move to the semantic phase. In this phase, a prompt generator in combination with an LLMC as well as an oracle is used once again. The prompt generator is fed with the (syntactically correct) candidates, a human-readable task description and unit tests. It will then generate different combinations of this data and feed them into the LLMC. All candidates of the LLMC can then be checked by the semantic oracle. In case of multiple valid candidates, token-based edit distance is used to select the one with the minimal number of needed changes [53]. A drawback of this approach is the need of an available large language model, which works as a black-box and can be expensive in use.

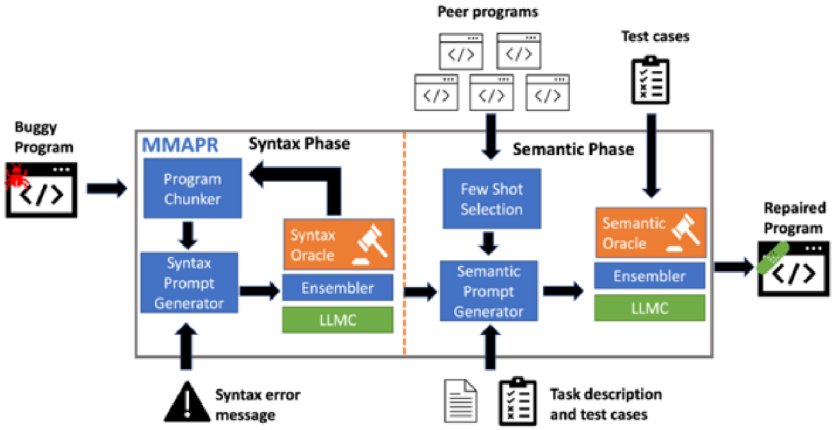


Figure 3.3: The overall procedure of MMAPR [53]

3.3 Effects of Feedback

By evaluating the results of student submissions in an undergraduate C++ course, Kyrilov and Noelle examined the usefullness of feedback generated by their automated assessment system. In this setup, students had to work on six C++ exercises during a three-hour long laboratory session and hand in their solutions to an automated assessment system, i.e., they had 30 minutes per exercise. This system then generated feedback in the form of "correct" or "incorrect" for a submission without further information. Analyzing the results of this study, Kyrilov and Noelle found out that only 43 % of the students with an incorrect first attempt where able to correct their error within five minutes. This led Kyrlov and Noelle to the assumption that a feedback of this kind is not enough of an aid for novices [20].

To provide learners with high-quality feedback, Kyrilov and Noelle clustered the incorrect submissions by error. A teacher could then provide high-quality feedback for each cluster instead of for each incorrect submission on its own. In their experiment, Kyrilov and Noelle achieved a number of clusters between 4 % and 31 % of the number of incorrect submissions, with the problem difficulty varying from calculating the area of a circle with a given radius (4 clusters per 100 submissions) to sorting an integer list (31 clusters per 100 submissions) [20].

Another approach of generating feedback by learning the different classes of errors and providing manual feedback for each class is presented in the form of the "Concepts and Skills based Feedback Generation Framework" by Haldeman et al. The aim is to "link errors to concepts and skills central to a programming assignment" [11] by manually generating hints for submissions that can then be used in future semesters. To be able to generate feedback, instructors have to create a test suite covering all aspects of the task. This test suite is then used to evaluate each student's submission to group them into clusters of the same failed tests. Instructors then have to update the test suite and re-run it until every cluster has the same logical error in all of its submissions. Once this is achieved, it is possible to manually write feedback for this error and return it to all submissions in the cluster. After applying the methodology to introductory course assignments, Haldeman et al. "found that the hints should provide useful feedback for the vast majority of incorrect submissions" [11].

PythonTA is a system designed for educational purposes to aid students with information about their code quality. Using static code analysis, PythonTA is able to check style errors, syntax errors or the correct usage of imports, e.g., forbidding students to import a specific module [22]. As shown by Liu et al., "students generally had positive impressions of PythonTA" [23]. This was detected by integrating PythonTA into a Python course with over 800 students. Nevertheless, students with no prior knowledge about Python programming had difficulties using PythonTA, which leads to the assumption that the integration into the course needs to be improved, especially for novices [23].

Chapter 4 Conceptual Approach

This chapter introduces the conceptual approach of Pyrat, the Python repair tool developed in this thesis. To structure the work and this thesis, we discuss the functional and non-functional requirements Pyrat needs to fulfill.

[R1] Python Language Features Support Existing tools like CLARA or Refactory do not support the complete Python language, e.g., Object Orientation is not supported [10, 14]. Section 4.1 analyzes three different introductory courses to check if there are Python features that are not supported by the presented existing code repair tools, solving Research Question 1.

[R2] Integrated Submission Validation Repair tools like CLARA or Refactory are only built to repair the buggy code and generate feedback for it, using a given set of correct submissions for the same problem. However, they do not include functionality to distinguish if a submission is correct or not and thus needs code repair. [10, 14]. As this Master's thesis aims to create a code repair tool that can be used by students while working on their exercise tasks, Pyrat should include functionality to verify if a student's submission is correct or not. Solely based on the result of these checks, i.e., without the need of interaction with a teacher, code repair should be executed. In section 4.2 we discuss the options for integrated submission validation.

[R3] Cold Start Code repair tools like CLARA depend on the availability of a large amount of reference solutions for a task to be able to fix buggy code [10]. For new tasks this can be a problem, as there are no correct student solutions known which could be used as reference solutions. Thus, Pyrat should offer options to easily create a sufficient number of reference solutions for the code repair. Section 4.3 discusses how these options can be designed.

[R4] Strategy Enforcement On occasion, educators may discourage students from employing the most straightforward strategy to a task. For instance, when introducing the concept of recursion, it can be beneficial to restrict students to the usage of recursion instead of looping structures in the corresponding exercises. Thus, Pyrat should be configureable in a way that submissions which do not work with the given strategy are marked as incorrect, even if the calculated result is correct. Furthermore, code repair should not generate feedback to fix the code in a direction that does not follow the requested strategy.

[R5] Verification of Non-Teacher-Authored Data Results obtained from Large Language Models (LLMs), e.g., generated reference solutions or repair suggestions, are not necessarily correct [18, 6]. The same applies for technically correct submissions for a task by other students. Even if all configured checks are passed this does not mean that a solutions is indeed correct and of good quality, mostly because of incomplete checks by the teacher [1]. This leads to the possibility of incorrect feedback when using erroneous LLM-generated or student-written codes as reference solutions for a task. To prevent this, Pyrat should offer the option to use external code only after it has been reviewed by a teacher.

[R6] Integration and Usability Each task that students can work on and generate feedback for in Pyrat needs to be configured in the system. To make this task easier for teachers, Pyrat should provide a graphical interface to add and configure tasks, including the validation checks and code repair settings. Nevertheless, the utilization of Pyrat should not be confined to its graphical interface, particularly with regard to the student interface that performs the code verification and repair functions. Thus, all functions of Pyrat should as well be accessible by a documented Application Programming Interface (API).

[R7] Scalability Scalability is one reason code repair tools are developed – the higher a number of students enrolled in a programming course gets, the more effort it is for teachers to provide manual feedback. Code repair algorithms need some time to generate feedback for a buggy solution [10, 14]. However, feedback is most effective if students receive it shortly after submitting their code [51]. While this is not a problem for courses where only a small number of submissions are made per hour, large courses like Massive Open Online Courses (MOOCs) come with an equally large number of submissions. Thus, Pyrat should be able to generate feedback in a reasonable amount of time after the student submitted their code and deliver said feedback. Furthermore, the technical setup of Pyrat should be designed in a way that can be scaled vertically for large courses if the need arises.

4.1 Python Language Features

The goal of this section is to find an answer for the first research question, i.e., which Python features have to be supported by a code repair tool. For this, we analyze three different courses; one of which is a university course, while the other two are online courses.

1. Lecture "Introduction into Programming for Data-Driven Sciences", RWTH Aachen [1]

2. Online Course "LearnPython.org", LearnX [2]

[1] https://learntech.rwth-aachen.de/go/id/jzoi, `https://learntech.rwth-aachen.de/go/id/jzoi`, last accessed 04/12/2024

[2] Learn Python - Free Interactive Python Tutorial, `https://learnpython.org/`, last accessed 04/12/2024

3. Online Course "Python", Google Education [3]

Based on these courses, a tool should support at least the Python concepts listed below.

A: Input and Output (1, 2, 3) In Python, input and output for command-line scripts is done using the built-in functions `input` and `print` [34]. While the print function is non-blocking, the input function waits until an input terminated by the newline character is provided [34]. This can be a problem for code repair tools relying on executing the code as they need to detect if the code is waiting for input and, if so, provide it.

B: Code execution without function definitions (1, 2, 3) In Java, one of the most used languages in MOOCs [25] and introductory progamming courses [8], a code file always needs to define a class with a static main method to be executable [28]. However, Python does not require the usage of classes or even functions, making the code in listing 4.1 a valid Python program [37]. Thus, courses can make use of this and introduce the most basic programming concepts like variable assignments before introducing the concept of functions or object orientation. As students can write buggy code for those tasks, too, code repair tools need to work with code that is not encapsulated in functions. This is currently not the case for the tools CLARA and Refactory [10, 14].

```
1 a = int(input("Your number a: "))
2 b = int(input("Your number b: "))
3 print(f"The sum of your numbers is {a + b}")
```

Listing 4.1: Valid Python code without classes and functions

C: Usage of built-in function names like `len`, `sum`, `round`, `min`, `max` (1, 2, 3) In Python, many functions like calculating the length of a string or list are performed by top-level built-in functions instead of public methods of the string or list [34]. As they are necessary for many programs, code repair tools need to support their use.

D: Usage of looping structures `for` and `while` (1, 2, 3) Python supports two different loop types: a for and a while loop [38]. While the while loop repeats as long as a condition is true, the for loop iterates over an iterable object defined beforehand [38]. While everything that can be computed using a for loop can also be done using a while loop, using a for loop can reduce the complexity of the code, making both loop types important for programming education. Both loop types are supported by all code repair tools listed in chapter 3. However, problems can occur if a buggy code contains infinite while loops if the code is executed.

E: Usage of conditional statements `if` and `else` (1, 2, 3) Another important control statement is the if statement used to conditionally execute parts of the code [38]. If statements are supported by all code repair tools listed in chapter 3.

[3] Google's Python Class | Python Education | Google for Developers, `https://developers.google.com/edu/python`, last accessed 04/12/2024

F: Function definitions (1, 2, 3) Functions can be used to structure code and improve reusability. In Python, functions can receive both positional and keyword arguments (and arguments that can be used as positional and keyword argument simultaneously) [38]. While CLARA and Refactory support functions in genereal, it is not possible to define functions accepting an arbitrary number of arguments with CLARA [10, 14].

G: Import and usage of external modules (1, 2, 3) Another way of code reusability in Python is the usage of external modules. Imported modules can either be part of the standard library or be manually installed, e.g, using a package manager like pip [39, 35]. While the behavior of modules from the standard library is known at the time a code repair tool is developed, this is not the case for installable modules. Thus, CLARA restricts the use of imported modules to a small subset of Python's standard library [10].

H: File access (1, 3) While file access is in general nothing else than working with with special functions, these functions come with side effects on the operating system level. Besides security concerns, code repair tools need to make sure that the assumptions a code makes about the directory structure, e.g., the existence of a file that is required to solve the task, are fulfilled, if the repair tool relies on executing the code. File access is possible when repairing code with CLARA or Refactory, however, it will lead to problems if files that are accessed by the buggy code or a reference solution are not created and prepared by the teacher. Furthermore, CLARA does not support the use of context managers, i.e., the `with` statement, which is the preferred way of dealing with file access in Python [10, 14, 37].

I: Object orientation (1, 2) "Object-oriented programming is central to modern software development and therefore integral to a computer science curriculum" [9]. Thus, a code repair tool needs to be able to fix code that contains object orientation. However, CLARA and Refactory both do not support the use of object orientation [10, 14].

J: Working with / intentionally raising exceptions (1, 2, 3) Exceptions in Python can be used to stop the program execution if something unwanted happens, e.g., when a string containing non-numeric characters is converted to an integer. As students tend to make mistakes when learning a new programming language, they are propably used to the behavior of a raised exception. Using try and except statements makes it possible to define fallback plans in case an exception is raised [38]. However, CLARA and Refactory cannot work with intentionally raised exceptions, i.e., code for a task that needs to raise an acception to be considered as correct cannot be repaired as they consider the raised exception as a failure [10, 14].

K: Recursion (1) As recursion is based on defining and calling functions (F), code repair tools that can work with function definitions can also work with recursion. However, infinite recursion can lead to the same problems as an infinite while loop.

L: Forcing special paradigms like using `lambda` or `list comprehension` (1) In programming education, teachers might want to force students to use special programming concepts, i.e., to use recursion instead of loops. This is not directly part of code repair, as a code repair tool can work with these tasks as long as it supports the concepts that are required to use. Nevertheless, teachers need to check if the concept that should be used was indeed used by the student. Furthermore, consider an example where students should define a recursive function which calculates the cross sum of a positive integer. The easiest fix for the buggy code in listing 4.2 is to replace the float division assignment (`/=`) in line 5 with an integer division assignment (`//=`). However, this would still not solve the task as the student is required to use recursion instead of a while loop. Thus, code repair tools need to be somehow aware of which concepts can be used to fix a code.

```
1 def crosssum(n):
2     res = 0
3     while n > 0:
4         res += n % 10
5         n /= 10
6     return res
```

Listing 4.2: Buggy Python code calculating an integer's cross sum using a while loop

M: Randomness (1, 2) Refactory uses unit tests to determine if a repair suggestion is indeed correct. These unit tests are simple input-output tests, i.e., teachers have to provide a set of Python expressions calling the function to be tested and the according return value as string. This leads to problems for tasks that accepts a range of correct answers, e.g., a random number between 0 and 50, as this cannot be checked by string comparison [14]. The same applies for the variable bijection matching in CLARA. Two variables are only considered as equivalent if they take the same values in the same order, which again is not the case if randomness is used [10]. Pyrat thus should be able to work with randomness, i.e., the fact that variable and return values can differ for two executions of the exact same code and that both results can be correct.

With the information derived from the courses, we find an answer to Research Question 1.

4.2 Submission Validation

Submission validation is needed in Pyrat to check if a student's submission contains errors that need to be repaired or not. One option for validating the correctness of student submissions is using unit tests, defined as the individual testing of software units by IEEE [15]. As teachers have no knowledge about the internal logic of a student's submission, the black box testing technique is ideal for validating the submission [17].

This concept is used in autograding tools like nbgrader [16] or AutoGrader [54]. In the example of nbgrader, teachers can add hidden test cells to the notebook and assign points to it. In the correction phase, every cell of the notebook is executed, including the hidden test cells. If and only if the test cell does not raise an exception, it is marked as passed and the points are granted to the student. In general, a task can be considered as correct if all test cells are passed.

For Pyrat, we introduce checks, a unit-test-like approach for submission validation. In general, Pyrat checks can have one of three types: *formal*, *semantic* and *warning*. Formal checks are used to check the submission for syntax errors. Another use case for formal checks is checking compliance with constraints that are not absolutely necessary for the semantic correctness of the code. This could entail the imposition of constraints on the utilization of programming concepts such as recursion, the prohibition of the use of external modules, or the verification of the existence of specific variables.

Semantic checks, on the other hand, are used to check whether a formally correct code calculates the correct results, i.e., is semantically correct.

Finally, it is possible to add warning checks to a task. These checks inform students about any aspect of the code that is not a direct mistake but could potentially harm readability or lead to future errors, e.g., using `list` or `int` as variable names.

Besides the check type, teachers can add a title for internal identification and feedback that gets delivered to the student should the check fail.

The most basic check available in Pyrat is the unit test. Here, the teacher can define two Python codes, a prefix and a suffix. When the check is executed, a Python file is created where the student's submission is placed between the prefix and the suffix. The check is evaluated as passed if no exception is raised, i.e., the Python interpreter exits with code 0. This extendable execution of unit tests allows teachers to decide whether or not to use frameworks like unittest [41] or pytest [19]. By default, the file is run without additional arguments. However, teachers can define a custom call signature to run the Python file.

To ease the creation of new tasks, Pyrat provides a number of predefined checks for recurring tasks using the libraries PyCheckMate[4] [3] and PythonTA[5] [23] that require little to no configuration. PyCheckMate focuses on checks that can primarily be assigned to the group of formal checks, including checks for compilation, the existence of a variable, function or class as well as for concepts like recursion, list comprehension or specific loops [3]. PythonTA, on the other hand, is designed to provide information about code quality and "support learners by identifying logical errors" [23]. This includes checks for the reuse of built-in names, unused imports or variables or the usage of undefined variables, all of them fitting in the category of warning checks. Even if most of the checks provided by PyCheckMate and PythonTa fit into the category of formal or warning checks as described above, teachers are free to assign another check type if they want.

Finally, it should be noted that checks can be assigned dependencies. If check B depends on check A, it is only executed if check A passed. By this, unnecessary check execution can be prevented, e.g., checks that rely on syntactically correct code are not

[4] pycheckmate · PyPI, `https://pypi.org/project/pycheckmate/`, last accessed 07/25/2024
[5] python-ta · PyPIhttps://pypi.org/project/python-ta/, last accessed 07/25/2024

executed if the code contains syntax errors. Also, checks can be stored as templates, making it easy to reuse checks that are required often in the same or a similar way.

4.3 Cold Start Problem

Upon the creation of a new task, teachers are required to provide detailed information to enable Pyrat to generate feedback for student submissions. Besides configuring the checks used for the submission validation, reference solutions need to be added as a base for the code repair and feedback generation.

As creating unit tests and reference solutions by hand is a time consuming process, we discuss the integration of LLMs for this task. A prominent example for an LLM is ChatGPT, released in 2022, which gathered more than 100 million active users in the first two months of its public availability [13]. Accessible by an API, it is possible to integrate ChatGPT into Pyrat in a way that teachers do not have to deal with the LLM directly [27].

To ensure R5, the verification of non-teacher-authored data, teachers have to review the generated checks and reference solutions before they are applied to the task. Details on the generation of checks are given in section 5.1.1, the generation of reference solutions is handled in section 5.2.

Once the first correct codes are submitted to Pyrat, they can be used in the repair process as well. However, correct submissions can be very similar especially for small and simple tasks [49]. This makes it inefficient to store every correct submission that has been made to Pyrat. To solve this problem, section 5.4 describes a clustering process to detect equivalent solutions, storing only one code of each equivalence set. With the generation of checks and especially reference solutions, we solve Research Question 2.

4.4 Feedback Generation

According to Shute, "Feedback used in educational contexts is generally regarded as crucial to improving knowledge and skill acquisition" [43]. Following Rivers and Koedinger, "students who receive help while programming do better in their courses" [42]. A simple feedback in the form of passed or not passed for a set of unit tests is thereby only helpful to a part of the students [20]. Thus, the tool developed in this thesis should not only detect the mistakes a student made for an exercise, but instead generate feedback that can be handed out to the student as an aid to find and fix the error.

Both CLARA and Refactory do not focus on feedback [10, 14]. As an example, see the buggy submission in listing 4.3. The task was to create a function `add_iterable` which returns 0 in case a dictionary is given as the only argument. If a list or a set is given, the function should return the sum of the values in the iterable object. The error in listing 4.3 is that the student uses a print call instead of a return statement.

Listing 4.4 shows the repair suggestion offered by Refactory based on another student's correct submission. While the suggestion is correct, it does not highlight the change that needs to be made in order to turn the buggy program into a correct one. This violates the third rule of Shute's "Formative Feedback Guidelines to Enhance Learning", requesting to "present elaborated feedback in manageable units" [43].

```
def add_iterable (iter):
    result = 0
    for element in iter:
        result += element
    if type(iter) == dict:
        result = 0
    print(result)
```

Listing 4.3: A buggy submission for the add_iterable Problem

```
def add_iterable(iter):
    result = 0
    for element in iter:
        result += element
    if type(iter) == dict:
        result = 0
    return result
```

Listing 4.4: Repair suggestion of Refactory for listing 4.3

In comparison, listing 4.5 shows the feedback created by CLARA. While this describes the needed correction instead of providing a fixed code and thus highlights the needed change, one must know the output schema of CLARA to be able to understand this output.

```
* Change '$ret := $ret' to '$ret := result' *after*
    the 'for' loop starting at line 3 (cost=1.0)
```

Listing 4.5: Repair suggestion of CLARA for listing 4.3

4.5 Technical Setup

The main technical goal for Pyrat is that it can be easily set up and does not require complex or even conflicting dependencies on its host system. This can be achieved using Docker[6], a virtualization system that builds containers for applications than can be executed on most systems as long as the Docker environment is installed. Thus, side effects with other software are prevented and updates with changing dependencies are easy to apply.

As a Docker container usually should execute just one task, Pyrat is divided into three Docker containers, as illustrated in figure 4.1. Thus, different dependencies of the various parts of Pyrat can be handled and security issues are reduced.

The management container serves as the interface between Pyrat and the users, both teachers and students. Built with the Next.js framework[7], it offers an API with Create, Read, Update and Delete (CRUD) access to all data that is stored in the storage container. Furthermore, a compact web interface is provided to assist in the administration of tasks, with a dedicated interface for students also available. Both the API and the web interface are described in section 5.5.

[6] Docker: Accelerated Container Application Development, https://www.docker.com/, last accessed 07/24/2024

[7] Next.js by Vercel - The React Framework, https://nextjs.dev, last accessed 07/24/2024

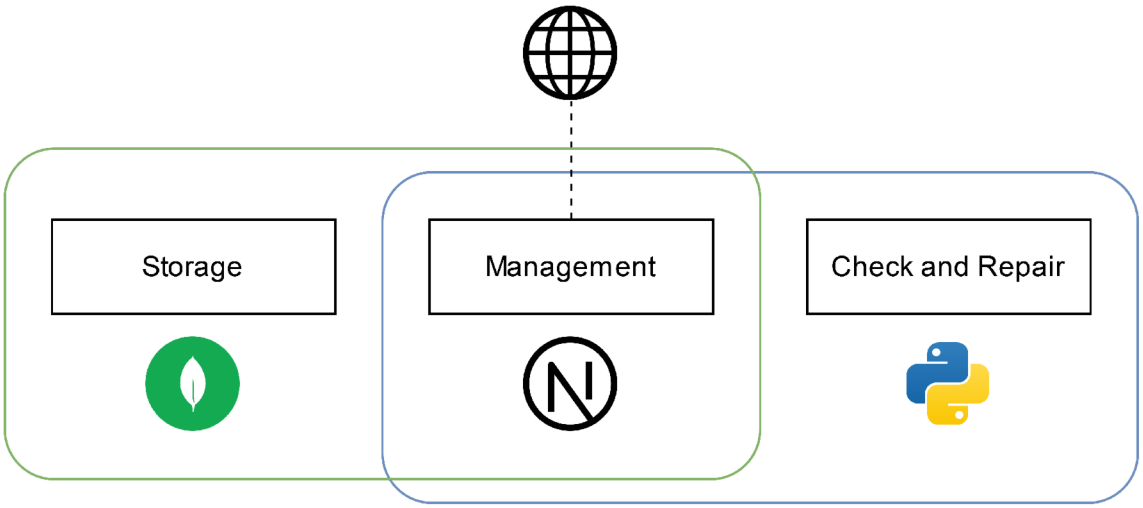


Figure 4.1: Docker containers and their networking

The storage container hosts a MongoDB database storing every information about the available tasks, including their checks and reference solutions. Section 5.6 describes the data format in detail.

The check and repair container hosts a Python environment used for executing the checks and repairs as well as some auxiliary functions to the web interface. To make the services of the check and repair container available via network, FastAPI is used. FastAPI is a small framework to easily create APIs using Python decorators and functions. [46]

For security reasons, communication between containers is only possible where necessary. As figure 4.1 illustrates, one docker network connects the management container with the storage container. Thus, the Next.js application is able to read and edit the tasks stored in the system. Another network connects the management container with the check and repair container, which is required to initiate the check and repair process from the web interface or the public API. As the check and repair container has no access to the storage container, it is necessary to first load every information, i.e., the check definitions and all reference solutions for a task, from the storage container and to forward them to the check and repair container afterwards. While this introduces a small overhead, it eliminates the need of database access in the check and repair container, where student code is executed and thus vulnerabilities might exist. Both the storage and the check and repair containers are not connected to the outer network, removing the need of authentication and authorization for both containers.

Lastly, this setup offers scalability. The check and repair container, which is the container with the highest computational load, runs completely stateless by getting all the necessary information in its requests. Thus, horizontal scaling techniques like offered from Kubernetes[8] can be used to distribute the computing load to multiple physical servers. The same applies for the management container built with the Next.js framework. Storing all data in the storage container and not in the runtime environment, horizontal scaling can be used for the management container as well.

[8] Kubernetes, `https://kubernetes.io/`, last accessed 07/24/2024

Additionally, Next.js is built to be executed on Vercels Edge Network[9] or using serverless Node.js functions, making it possible to host the web interface on highly scalable content delivery networks. While the storage is naturally not stateless, MongoDB offers scaling options using replication. This highly increases the number of possible read operations as they can be handled by every replication node of a replica set. As write operations, i.e., operations that can only be performed by the primary node, only occur if teachers make changes to the task or a new cluster of correct student submissions is detected, scalability will be no problem for usual course sizes. Otherwise, partitioning of the data can be used to further reduce the load on individual containers by having multiple replica sets with own primary nodes. [26]

As every task (including its checks, reference solutions and correct student submissions) can be administrated using the API, integration of Pyrat into existing exercise or programming environments is possible as long as these environments can be extended, e.g., by plugins. This solves the third Research Question. As an example, it is possible to create an extension for Jupyter Lab that integrates Pyrat into the notebooks [31].

[9] Edge Network, `https://vercel.com/docs/edge-network/overview`, last accessed 07/24/2024

Chapter 5 Implementation

This chapter introduces the implementation of Pyrat. Figure 5.1 illustrates a high-level overview of the task workflow from the teacher's perspective, featuring the different parts of the implementation.

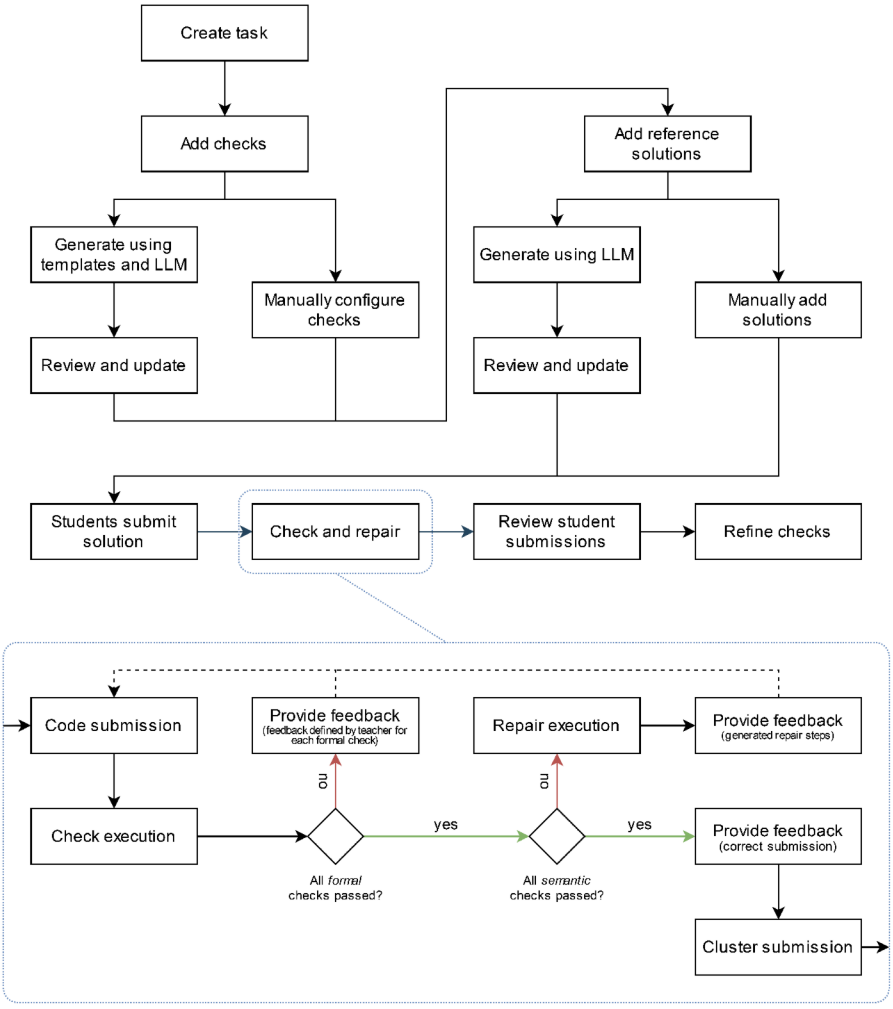


Figure 5.1: High-level overview of the task workflow from the teacher's perspective

First, a teacher needs to create a new task and provide general information, most importantly a human-readable description that tells the students how to solve the task.

Section 5.1 describes the process of generating or manually adding checks that are executed on code submissions to test if the submission is correct or not. Section 5.2 describes how a Large Language Model (LLM) can be used to add reference solutions as a basis for the code repair.

Once a task is created, students can work on the task and submit their codes. After executing the checks, Pyrat performs the code repair (section 5.3) in case the submission is not correct. Feedback can then be used by students to fix errors in their code, either leading to correct code or to new feedback. Using clustering of correct submissions (section 5.4), teachers can enhance the set of reference solutions used for repair.

Section 5.5 describes the web application that can be used to create and edit tasks including their checks and reference solutions as well as to run the student submissions.

5.1 Checks

As shown in figure 5.1, check execution is the first thing that happens after a student submits their code. By executing the checks, Pyrat can ascertain whether code repair is necessary for the submission.

5.1.1 Check Generation

Because it is complicated and time-consuming to create the necessary checks by hand, the system aims to facilitate this task by using an LLM to extract essential information from the task description and build the checks based on this information.

As PyCheckMate [3] and PythonTA [22] are included as libraries to remove the need of writing unit test code for boilerplate checks, the LLM should be used to create the checks using PyCheckMate and PythonTA. When using LLMs, prompt engineering is important to get high-quality results [50]. To build a guideline for prompt engineering, White et al. developed prompt patterns. Following White et al., "prompt patterns provide a codified approach to customizing the output and interactions of LLMs" [50]. The first approach to generate these checks is to prompt the LLM for every available check with the prompt in listing 5.1 based on White et al.'s Template Pattern [50]. While the first sentence introduces the LLM to the overall context, the second one instructs the model to use JavaScript Object Notation (JSON) mode as output and is a necessary message if the output should be valid JSON [27].

```
$ You are an assistant creating unit tests for Python introductory problems.
$ Respond with a JSON object containing a boolean value 'check_needed' and,
    if needed, an array 'args' with the arguments for the check. Each argument
    should be an object with the keys 'arg' and 'value'.
$ <A description of the check including the required and optional args>
$ <Task description created by the teacher>
```

Listing 5.1: Prompt set 1 for check generation

As an example, we built the prompt for a PyCheckMate function checking if a given built-in function is used in the student's submission. This check should be created for a task where the student is required to define a checksum function calculating the cross sum of a number. Adding the check and task descriptions to the prompt set in listing 5.1, this results in the prompt in listing 5.2. When executing this prompt on the ChatGPT-3.5-Turbo model, we get the result in listing 5.3.

```
$ You are an assistant creating unit tests for Python introductory problems.
$ Respond with a JSON object containing a boolean value 'check_needed' and, if
    needed, an array 'args' with the arguments for the check. Each argument
    should be an object with the keys 'arg' and 'value'.
$ Add this check for each built-in function that is required by the task
    description. Use the following arguments: function_to_use: str, name of
    the function that is required.
$ Implement a function `checksum(number)`. Your function should calculate the
    cross sum of the passed number, i.e., the checksum of a number is the sum
    of its digits. For example, the checksum of the number 721365 is
    7+2+1+3+6+5 = 24. Hint: Only positive integers (>=0) are passed to the
    function.
```

Listing 5.2: Prompt for the built-in function check for the checksum task, built on listing 5.1

```
{
  "check_needed": true,
}
```

Listing 5.3: ChatGPT-3.5-Turbo result for the prompt in listing 5.2

Running the prompt set from listing 5.1 for all available checks for the checksum task, the LLM suggests the following checks. For easier readability we provide a summary of the responses instead of the JSON objects.

1. The built-in function `str` has to be used.
2. The code should define a class `checksum` with one parameter.
3. The code should define a function `checksum` with one parameter.
4. The code should use a `while` loop.
5. The code should use a `for` loop.
6. The code should define a lambda function `checksum` with one parameter.
7. The code should use a list comprehension.
8. The function `checksum` should be defined recursively.

Based on the task description, some of the checks above do not make sense, e.g., the description says nothing about defining a class (2) or using lambda functions (6) or list comprehensions (7). Also, it is not a requirement to define the `checksum` function recursively (8), but if doing so, the usage of `for` (5) and `while` (4) loops seems inconsistent. The check creation behavior was similar for other tasks, leading to the assumption that check creation using an LLM with this prompt set is not useful.

The second approach, again using the Template Pattern, uses the LLM just to extract the necessary information out of the task description. Then, a deterministic procedure builds the tests based on this information. As a positive side effect this reduces the number of requests to the LLM from one per check to one in total. Listing 5.4 shows the prompt to the LLM, again following White et al.'s Template Pattern. A JSON object describing both the output scheme and the needed information is sent to the LLM with the task description. In a second step, this extracted information is then used to automatically create the requested checks.

```
$ You are an assistant extracting relevant information for unit tests for
    Python introductory problems.
$ Respond with a JSON object. You can use the following template, replace the
    description with the relevant information.
$ {
  "builtin": "If it is obligatory to use built-in function, insert an array of
      their names, else leave this option.",
  "functionName": "If a function is needed, insert the function name here,
      else leave this option.",
  "functionParamsCount": "If a function is needed, insert the number of
      positional parameters here, else leave this option.",
  "functionParamsArgs": "If a function is needed and should accept an
      arbitrary number of positional arguments, insert true here, else leave
      this option.",
  "functionParamsKwargs": "If a function is needed and should accept an
      arbitrary number of keyword arguments, insert true here, else leave
      this option.",
  "while": "If it is obligatory to use a while loop, insert true here, else
      leave this option.",
  "for": "If it is obligatory to use a for loop, insert true here, else leave
      this option.",
  "tryExcept": "If it is obligatory to use a try-except block, insert true
      here, else leave this option.",
  "recursive": "If the function should be recursive, insert true here, else
      leave this option.",
  "imports": "If it is obligatory to use imports, insert an array of their
      names, else leave this option.",
}
$ <Task description created by the teacher>
```

Listing 5.4: Prompt set 2 for check generation

This time, the LLM suggests to create one check for the function name (it should be `checksum`) and one for the number of positional arguments (it should be one), which is everything that can be derived from the short task description. Pyrat then automatically creates these checks for the teacher.

Code Convention Violation Hints Besides generating the checks for formal alignment to the task, the teacher can add a number of checks for code convention violations by one click. As code convention does not depend on the task description, no LLM integration to extract information is used here. Instead, a pre-defined set of nine checks is added. In the default case, these checks do not mark a student's solution as failed, but give a hint instead, i.e., they are of type *warning*. However, teachers are

free to select another check type after the checks are created and also remove checks they do not want to use. Following checks are added to the task:

- Usage of top-level code (should be removed if solution should contain top-level code)
- Usage of global variables
- Usage of unused imports
- Definition of unused variable names
- Usage of undefined variables
- Usage of the loop variable (in for-loops) outside the loop (which is possible in Python but might introduce logical errors)
- Usage of loop keywords `break` and `continue` outside of loops
- Usage of the `return` statement outside of functions
- Existence of unreachable code

5.1.2 Check Execution

When executing the checks, security concerns have to be taken into account. While checks such as those offered by PyCheckMate[1] and PythonTA[2] are static checks and thus do not rely on execution of the student's potentially harmful code, unit tests that check for the correct result for a given input must execute the tests [3, 22]. In addition to malicious attacks, unit tests might harm the system in an unintended manner, e.g., blocking the server with accidental infinity loops or overwriting important test files or system resources.

To provide a secure environment for code execution, the docker container creates a low-privileged user *tester* with an own home directory. On unit test execution, a temporary subdirectory in this home directory is created and the code is placed inside this directory. Using Python's *subprocess* module, the unit test is then executed under the *tester* user with a timeout configurable in the check definition. In case of a unit test failure, the output of the code is captured, enabling the teacher to see failure reasons when creating reference solutions and checking them [40, 5].

[1] pycheckmate · PyPI, `https://pypi.org/project/pycheckmate/`, last accessed 07/25/2024
[2] python-ta · PyPI, `https://pypi.org/project/python-ta/`, last accessed 07/25/2024

As checks executed by PyCheckMate and PythonTA are static, there is no necessity to set up the environment expected by the student's task, e.g., files to read from, before those checks are executed. However, this is not the case for unit tests executing the code to check the semantical correctness of the submission. As stated in section 4.2, unit tests consist of a prefix and a suffix code. Using the prefix code it is thus possible for the teacher to set up the testing environment in any required way. As both the prefix and the suffix code are combined with the student's submission in the same Python file, all of those code fragments are executed under the low-privileged *tester* user. Thus, setup of eventually needed files is only possible in the temporary directory created for this single check execution, i.e., using relative paths. An advantage of this is that all files that are created for the check execution are automatically removed after the execution, removing the need of manual garbage collection by the teacher. Also, this ensures that the execution of a check for a submission A cannot affect the check execution for submission B.

5.2 Solution Generation

Pyrat's code repair needs at least one reference solution to calculate differences and generate feedback, but especially for more complex tasks multiple solutions following different approaches improve the results (chapter 6). As writing multiple reference solutions is time-consuming for the teacher, an LLM should be integrated into Pyrat to ease this step.

For evaluation of the developed prompts, we use CodingBat[3] developed at Standford University, offering 72 exercises for Python novices. For each task, CodingBat provides a task title, a short description of usually less than five sentences, a function definition header and a few input/output examples. Besides a brief description of the task and the requested function name, every task comes with a set of unit tests executable by an Application Programming Interface (API). As an example, see the front_times task: *Given a string and a non-negative int n, we'll say that the front of the string is the first 3 chars, or whatever is there if the string is less than length 3. Return n copies of the front.* The calls in listing 5.5 are given as an additional aid for the students to self-check the code.

```
front_times('Chocolate', 2) -> 'ChoCho'
front_times('Chocolate', 3) -> 'ChoChoCho'
front_times('Abc', 3)       -> 'AbcAbcAbc'
```

Listing 5.5: Example input/output tests for the front_times task

A second source for evaluation is CodeCheck[4]. Similar to CodingBat, 165 tasks for Python novices are offered for self training. 153 of the tasks come with a description and the requested function name as well as a set of unit tests executable by an API. The remaining 12 tasks do not use the concept of functions, what makes it impractical to use the unit tests provided by CodeCheck. As an example, see the the following description: *Given a string s, return the string with adjacent duplicates removed. For example, Mississippi yields Misisipi. You may assume there is at least one character in the given string.*

[3] CodingBat Python, `https://codingbat.com/python`, last accessed 04/14/2024
[4] CodeCheck, `https://horstmann.com/codecheck`, last accessed 04/14/2024

The initial test uses the prompt in listing 5.6. As both CodingBat and CodeCheck do not include the expected function name in the description (it is shown in the code field instead), it is added as an extra message.

```
$ You are an assistant writing sample solutions for Python introductory
    problems.
$ Respond with a JSON object containing the code for the given problem as a
    string with the key 'solution'.
$ <Task description from CodingBat or CodeCheck>
$ The function should be named <funcName>.
```

Listing 5.6: Prompt set 1 for solution generation

Figure 5.2 shows the performance of the prompts for CodingBat and CodeCheck. Using these messages, ChatGPT fails to write fully working code for one CodingBat task. From the 47 generated codes from CodeCheck that failed completely, 22 failed because the function name was chosen incorrectly by ChatGPT. Another 20 codes failed because ChatGPT included example function calls with print statements into the solution, which breaks the unit tests. This is also the case for the one failing test from the CodingBat dataset.

To reduce the number of failed checks, an improved prompt set (listing 5.7) was designed. Figure 5.2 demonstrates that these prompts decrease the number of tasks with a complete unit test failure by 85 %, while the number of tasks with partial unit test success remains constant. Eight tasks from CodeChef achieve a better test score with the first prompt than they do with the second set. In total, only 25 tasks fail at least one unit test with both prompt sets, of which five fail completely.

```
$ You are an assistant writing sample solutions for Python introductory
    problems.
$ Respond with a JSON object containing the code for the given problem as a
    string with the key 'solution'.
$ Do not add exemplary tests to your code and do not call the print function.
$ <Task description from CodingBat or CodeCheck>
$ The function should be named exactly '<funcName>', do not change this name.
```

Listing 5.7: Prompt set 2 for solution generation

These results match those of Kiesler and Schiffner, where ChatGPT-3.5 was able to solve 94.4 % of the CodingBat tasks[18]. However, Kiesler and Schiffner do not use the API to ChatGPT, i.e., code has to be extracted manually by the study conductors instead of being automatically parsed. Denny et al., who used Codex, the GPT-3 based LLM used for code generation in Github's Copilot, solved 47.6 % of the tasks from CodeCheck with the initial prompt. Another 31.9 % of the tasks could be solved by manually editing the description of the task, which leads to a total of 79.5 % solved tasks. [6]. Even without manual editing of the prompts shown here, Prompt Set 2 slightly outperforms the results of Kiesler and Schiffner and Denny et al. with 100 % and 81.0 % solved tasks for CodingBat and CodeCheck, respectively. However, as Pyrat tasks do not have an extra data field to store an expected function name (it should be stated in the task description instead), we do not use the last sentence of the prompt set in listing 5.7 in the implementation.

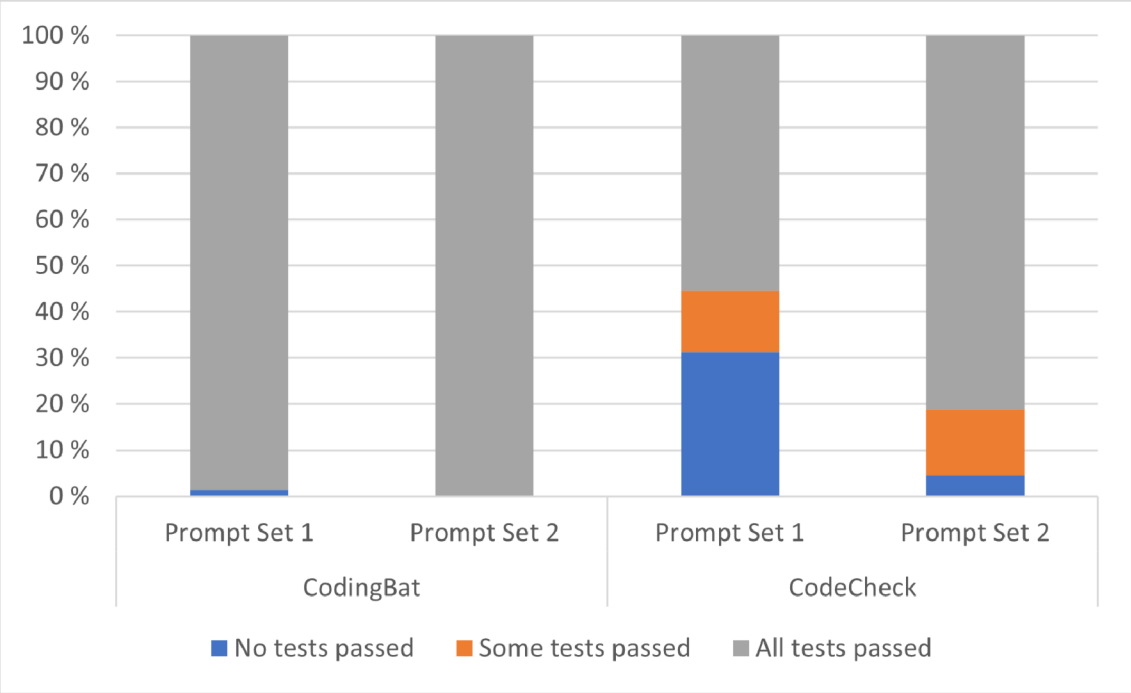


Figure 5.2: ChatGPT prompt evaluation results, relative values

The difference between the results of CodingBat and CodeCheck problems can be explained by more complex problems in the CodeCheck dataset.
To get valid reference solutions for every task, the following process is implemented into the teacher's interface of the code repair system. First, Prompt Set 2 (listing 5.7) is sent to the LLMs API, including the task description written for the students. The code generated by the LLM is then validated against all existing checks. In the next step, the teacher has the option to review the code as well as the validation result. If the code is not yet sufficient for a reference solution, the teacher can either edit the code manually or send a refined request to the LLM, containing the original messages, the wrong code generated by the LLM and the change query. In general, this concept can be categorized as a human-in-the-loop approach, often used in the context of machine learning [52]. Here, human interaction in the process of machine learning is used to train the model, i.e., generated results are reviewed and rated by a human.
The following example reusing the checksum problem from section 5.1.1 describes the benefits created by this human-in-the-loop concept. After sending the initial prompt, the code in listing 5.8 is returned by ChatGPT. While this code solves the given problem, casting the number to a string and each character back to a number could be considered as bad design. Thus, the following change request is sent to ChatGPT: *Do not cast the number to a string, use a mathematical approach.* This results in a new code with a better design, again solving the problem correctly (listing 5.9).

```
1 def checksum(number):
2     return sum(int(digit) for digit in str(number))
```

Listing 5.8: First result, generated by ChatGPT for the checksum problem using prompt set 2 (listing 5.7)

```python
def checksum(number):
    total = 0
    while number > 0:
        total += number % 10
        number //= 10
    return total
```

Listing 5.9: Second result, generated by ChatGPT for the checksum problem using prompt set 2 (listing 5.7) using a refinement statement

5.3 Code Repair

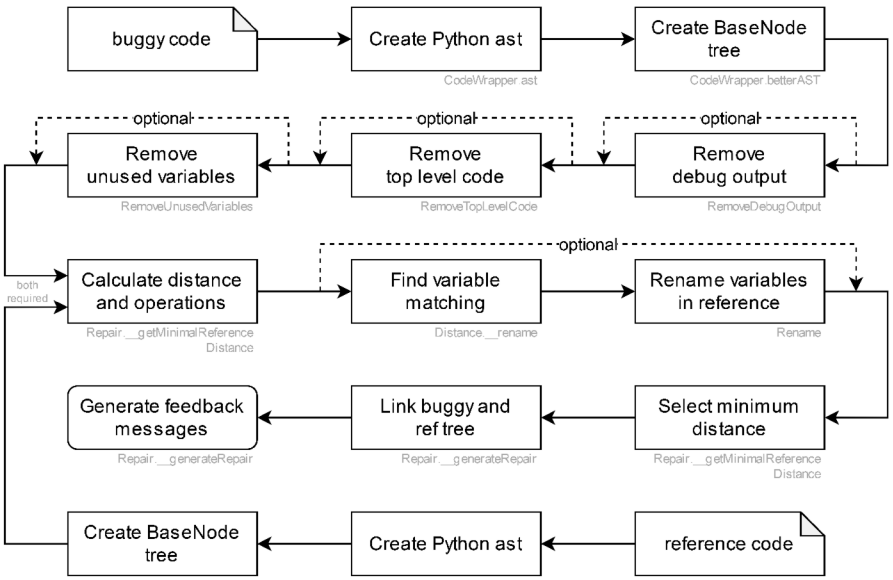


Figure 5.3: Overview of the repair algorithm implemented in Pyrat. Classes implementing the functionality are given below the boxes.

In case a submission is buggy, code repair needs to be applied to generate feedback on how to fix the code for the student. All of the repair tools described in chapter 3 have their downsides: CLARA and Refactory rely on code execution and thus have problems with semantic errors like infinity loops or side effects like file handling [10, 14], LLM-based approaches like MMAPR do not run deterministically [53]. Thus, figure 5.3 visualizes the repair algorithm developed for Pyrat, a static code repair algorithm that repairs buggy submissions without executing them. While code execution is in general still needed to check if the submission is correct or not (see section 5.1.2), a failure in unit tests simply declares the need for repair. It does not matter if the reason for this failure is an infinity loop triggering a timeout, a semantic error leading to incorrect output, or anything else.

To develop a code repair algorithm, some preliminaries have to be set. First, code repair should not run on a strictly syntactic level. While the Python Enhancement Proposal 8 (PEP 8) defines a style guide that should be followed when writing Python code [47], learners might not be aware of this. Also, even when following the style guide, there are still multiple syntactic expressions of the exact same semantics. A

trivial example for this is the semantic equivalence between single and double quotes in Python [47].

Second, all Python features described in section 4.1 should be supported by the algorithm. This applies in particular for features not yet supported by existing tools like CLARA or Refactory, i.e., code outside of functions, object orientation or exceptions.

Third, code repair should work also on code that is syntactically correct but not completely executable. A reason for this could be the implementation of an infinity loop or endless recursion. As described in section 3.2, this is not the case for existing tools that rely on executing the code as fully executing code with an infinity loop or similar is not possible for obvious reasons.

5.3.1 Creating the Abstract Syntax Tree and BaseNode Tree

In general, Pyrat's repair algorithm works by comparing a buggy student submission with one ore more correct reference solutions. To do this, it generates the Abstract Syntax Tree (AST) for each code snippet using Python's *ast*[5] module (section 2.1). While other third-party libraries with similar capabilities are available as well, using a built-in library that is used by the compiler comes with the benefit of guaranteed maintenance of the library, also for coming Python releases and new features.

Access to the most relevant information is given in the subclasses of the ast module. However, they are not sufficient to be used in the tree edit distance directly. As every Python element and feature has an own subclass in the ast, we need to define a cost metric that includes if an update between objects of two subclasses is possible or not (sections 5.3.3 and 5.3.4). With the data structure chosen for the ast module, accessing the elements of the tree is only possible top-down. While every element holds an ordered list of its children, these children have no link to their parent or sibling nodes. However, this information is required in the feedback generation process to describe the location of the error to the student (section 5.3.8).

To solve these problems, a new tree is constructed from the AST. As for the ast module, each type of node is represented by an own class with suitable attributes. If multiple AST nodes belong to a group, i.e., updates are possible between the different node types (section 5.3.4), they are represented by one class in the new tree.

As every node of the tree needs some methods and attributes in the same way, a base class `BaseNode`[6] is created that the different subclasses for the ast nodes can inherit from. First, this includes links to the parent, first child, predecessor and successor as well as methods to add a child or sibling to the node to construct the tree. As the feedback generation process will link two trees, i.e., the buggy tree and the reference tree based on the generated tree edit distance operations (section 5.3.7), a link to the equivalent node of the other tree is stored as well. To give students information on where the error in their code is, location information, i.e., start and end line of the element, is added to the node as well.

[5] ast - Abstract Syntax Trees - Python 3.12.4 documentation, `https://docs.python.org/3/library/ast.html`, last accessed 07/25/2024

[6] pyrat/src/base_repair/nodes.py, lines 19 - 501

A public method[7] is offered to calculate the update cost to another node. First, the method checks if both node objects are of the same class, i.e., if an update is possible between them. If this is the case, a protected abstract method[8] that has to be defined by each node class itself is called to get the correct value. Otherwise, twice the sum of insertion and removal cost of the two nodes is returned to make an update more expensive than a replacement. Using two methods, a public and a protected one, it is possible to define the exact cost metric in each node separately while not overwriting the check if both nodes are of the same class.

5.3.2 Repair Improvements

The created BaseNode tree now represents the logic of the submitted code exactly and can be used to calculate the distance to all reference solutions and to compute the necessary operations as described in section 5.3.3. However, some parts of the code, e.g., debug outputs, top-level code or unused variables are irrelevant for the output of the submission and thus for the repair [49].

Thus, Pyrat has the option to remove debug output[9]. However, not every print statement can be removed without side effects. As an example, consider an assignment where a function should be defined that takes a list as its only argument and returns its penultimate element. All side effects, e.g., outputs and changes to the input list, are not relevant for the assignment. While the code in listing 5.10 solves this assignment, removing the print statement would result in an incorrect code as the function would now return the last element.

```
1 def function(data):
2     print(data.pop())
3     return data.pop()
```

Listing 5.10: Example code of an output with side effects

Thus, it is necessary to only remove output statements that are guaranteed to be free of any side effects. As the repair algorithm is static, i.e., it cannot execute the code, we need to define a set of outputs that are guaranteed to be side-effect free in any case. Clearly function calls are not guaranteed to be side-effect free, so output statements that print the return value of a function cannot be removed. The same holds for subscription and operators, as they can be defined for custom objects however the author of an objects likes. Thus, Pyrat only removes print statements if all positional arguments, i.e., the arguments that are printed, are either constants or plain variable names. However, it should be noted that it is also possible to define the `__str__` method of a class in a way that it is not side-effect free even if this contradicts a good code style.

[7] `getUpdateCost`, pyrat/src/base_repair/nodes.py, lines 382 - 395
[8] `_getSelfUpdateCost`, pyrat/src/base_repair/nodes.py, lines 172 - 179
[9] pyrat/src/base_repair/transfomers/removeDebugOutput.py

Additionally, Pyrat introduces another transformer that removes unused variables[10]. The same constraints as above hold for variables, too – a variable can only be removed if this removal is side-effect free, i.e., a constant or another variable is assigned. Otherwise, it is possible to remove the assignment and add the value as an expression instead, e.g., replacing `myVar = myFunction()` with just `myFunction()`. However, this can lead to complicated feedback, e.g., if the expression has to be removed as part of the code repair. Then, the student will just receive a message to remove the function call instead of the complete variable assignment.
For assignments that require students to create a function and nothing else, it is also possible to remove all top level code from the code[11]. Thereby, we define top level code as every code that is not indented, e.g., not part of a function or class. The only exception are constant assignments, function and class definitions and imports [22]. This way, outputs that print the result of a function call, i.e., the call of the function that is to be defined by the student, can also be removed before clustering and thus from the later saved reference solution.

5.3.3 Calculate Distance and Compute Operations

To find the differences between the buggy code and a reference solution, and, if multiple reference solutions are available for a problem, to find the closest reference solution, the APTED algorithm is used (section 2.2). To calculate the tree edit distance between two trees, a cost metric is needed that assigns a deterministic cost to every possible operation, i.e., inserts, removes and updates.
APTED's default Levenshtein-based cost metric is not suitable for the difference between two ASTs. While each node in the tree has a label, e.g., Module, ListComp, Name or Pow, the string edit distance between them is not based on the effort a student needs to put into repairing their code following this update operation. As an example, the labels Is and If have a string edit distance of one, while Is and Lt (the less-than operator) have a distance of two. Nevertheless, it is easier for a student to exchange the `is` operator with a `<` operator than with an if-condition.
To make the APTED algorithm work with the ASTs, we need to define a cost metric that (a) assigns a valid cost for insertions and removals, (b) assigns a valid cost for updates and (c) defines between which elements an update is possible at all.
While the APTED algorithm in general uses updates instead of removing and re-inserting, as this is always cheaper due to the calculation of the Levenshtein distance [21], this is not possible for the feedback generation of Pyrat. Here, we use updates if and only if the change of a node of the buggy tree to a node of the reference tree can be described by a short description, e.g., changing the name of a variable or a constant value. In any other case, e.g., if the student uses an if-condition, but a while-loop has to be used instead, we use remove and insert operations. Thus, the feedback generation algorithm creates separate messages for the removal and the insertion instead of providing one update message. As both nodes have no similarity, it is easier to provide separate messages for the algorithm and also easier to understand for the student. Section 5.3.4 discusses between which AST nodes updates are possible.

[10] pyrat/src/base_repair/transfomers/removeUnusedVariable.py
[11] pyrat/src/base_repair/transfomers/removeTopLevelCode.py

For the insertion and removal (a), the cost is defined as the size of the subtree for every element. As an example, see the BinOp element in listing 5.11. It has three children: a name, the power operator and a constant value. While the power operator and the constant are leaves and thus have a size of 1, the name has a single child ctx and has size 2. In total, the BinOp element in listing 5.11 has a size of 5, which is used for insertion and removal.

```
Module(
body=[
  Expr(
    value=ListComp(
      elt=BinOp(
        left=Name(id='i', ctx=Load()),
        op=Pow(),
        right=Constant(value=2)),
      generators=[
        comprehension(
          target=Name(id='i', ctx=Store()),
          iter=Call(
            func=Name(id='range', ctx=Load()),
            args=[
              Constant(value=10)],
            keywords=[]),
          ifs=[],
          is_async=0)]))],
type_ignores=[])
```

Listing 5.11: Small ast example (revisiting listing 2.2)

The update cost for valid updates (b) is more individual due to the complexity of some AST nodes. In general, two constraints must hold for every update cost. First, the cost must be greater than zero to be noticed as an update. Second, it must be smaller than the sum of the removal cost of the node in the buggy tree and the insertion cost of the node in the reference tree. By this constraint it is ensured that updating the node is cheaper than removing and re-inserting it with some values changed, which would lead to more complex repair instructions.
Using the APTED library, it is not possible to define that updates between certain types of nodes are not possible (c). As an alternative, we assign costs of two times the sum of removal and insertion to all update operations that are not allowed. This ensures that removing and installing is cheaper than the prohibited update.

5.3.4 Possible Updates

In general, updates are only possible between the same node type, e.g., a name can be updated to another name, but not to a function definition. While an update does not change the tree in a different way then removing the old node and creating a new one, this difference is important for the feedback generation (section 5.3.8). Nevertheless, some leaf nodes are grouped together as listed below. Nodes belonging to the same group can be exchanged at a lower cost of usually one, as changing them can be explained in a simple sentence and does not require multiple steps of work for the student.

- Binary operations like `+`, `-` and bitwise operations
- Unary operations `not`, `~`, `+` and `-`
- Boolean operations `and` and `or`
- Comparators for equality, less/greater than, `is` and `in`
- Loop keywords `break` and `continue`
- List-like structures `list`, `set` and `tuple`
- Comprehensions of `list`, `set`, `tuple` and `dict`
- `global` and `nonlocal statement`

The `async` keyword can be added to or removed from function definitions and for-loops with a cost of one, even if the async and non-async variant are represented by own nodes in the ast module. Another special case is the Expr node. It is used as a wrapper for statements that appear by themselves if the return value is not used, e.g., a function call without a variable assignment. As an expression has no syntax that students can add or remove and an Expr node always has exactly one child, the child is directly added to the tree instead.

For all other nodes an update cost is calculated only for updates between the same node type. While some nodes need a more complex comparison logic, e.g., a function definition needs to check for changes in the arguments or decorators, others like a while loop need no comparison logic at all since all update costs are calculated on child level.

5.3.5 Variable Matching and Renaming

The repair algorithm presented in this section works by making the buggy solution equivalent to a known reference solution. While this will always result in a correct code, it might introduce unnecessary feedback messages as parts of the reference solution can be changed without altering the code's behavior. A large source of unnecessary feedback messages are differences between the naming of variables in the buggy and the reference solution. As an example, see the buggy code and the corresponding reference code in listings 5.12 and 5.13, respectively. While the solely error is that the student multiplied the radius with three instead of two to get the diameter for the circumference calculation, additional feedback is generated. Two feedback messages tell the student to replace the variable *rad* with *radius*, once in the function argument and once in the calculation in the second line. Another two messages tell the student to replace the variable *res* with *result*, once in the assignment in the second line and once in the return statement. However, as they are all local variables of the function, their naming is irrelevant for the semantics of the code.

```
def circumference(rad):
    res = 3 * rad * 3.141
    return res
```

Listing 5.12: Buggy code for circle circumference calculation

```
def circumference(radius):
    result = 2 * radius * 3.141
    return result
```

Listing 5.13: Reference code for circle circumference calculation

To remove these feedback messages, the repair algorithm needs to find a match between variables that are used in a similar, but not necessarily equivalent way. In their paper and the included tool CLARA, Gulwani et al. trace the values of a variable over the execution on a given input to find a variable bijection between both codes [10]. Similar is done by Hu et al. for Refactory [14]. However, both algorithms rely on executing the codes on a predefined input. As the repair algorithm of Pyrat should run statically, this is not an option.

Another possible option is presented by Vogt in his analysis of programming equivalence. To ignore the naming of variables when checking if two programs are equivalent, Vogt's algorithm renames all variable names following a predefined scheme, i.e., in numeric order [49]. However, this algorithm relies on generating the symbol tables to calculate the scope of every identifier in the code. As the line numbers in the symbol tables and the AST have to match, it is not possible to apply this algorithm if at least one of the improvements in section 5.3.2 has been executed before [49]. Thus, we develop an own algorithm for Pyrat's code repair to match different variable names between the buggy and the reference solution.

With Python's ast module, it is possible to retrieve a list of all variable definitions and usages. Using this information and the repair operations generated by the APTED algorithm, we can find variable names that match without executing the code.

As a preliminary we have to define which ast nodes are names that can be updated. In the implementation, we use an interface class `NameLike` that enables access to the name of such nodes. The `ast.Name` subclass "can be used to load the value of a variable, to assign a new value to it, or to delete it" [33]. Class and function definitions, represented by the subclasses `ast.ClassDef` and `ast.FunctionDef`, also have names. When a class or function is called, the `ast.Name` subclass is used in the tree, making it hard to determine if the node refers to a variable or to a class or function definition. Thus, we include class and function definitions as well by making their BaseNode subclasses also subclasses of the `NameLike` interface. Lastly, functions can also get arguments that are not represented as name nodes, thus they inherit from the `NameLike` interface as well.

The idea of the algorithm in Pyrat is to find a bijection of variables by iterating over the computed list of operations for the buggy code against a reference solution. Listing 5.14 shows the used algorithm as pseudocode.

In lines one and two, the algorithm instantiates two empty mappings, one mapping variables from the buggy code to their matching variables in the reference solutions and one mapping vice versa. The algorithm then iterates over all pairs of name-like nodes in the list of operations. If the pair is not yet part of the mapping, it is added to both lists. If a name from the buggy code maps to more than one name in the reference code, e.g., one feedback message replaces name x by y and another one replaces x by z, it cannot be part of a variable bijection. Thus, lines eight and nine remove all mappings where a name maps to more than one name. Last thing is to

derive a bijection by selecting all names b from the reference solution that map to a name a in the buggy code that maps back to b.

```
buggyToRef = new mapping
refToBuggy = new mapping

for NameLike a, NameLike b in operations:
    buggyToRef[a.name] &= b.name
    refToBuggy[b.name] &= a.name

buggyToRef = {k: v[0] if v has exactly one element}
refToBuggy = {k: v[0] if v has exactly one element}

bijection = {b: a for b, a in refToBuggy if buggyToRef[a] == b}
```

Listing 5.14: Pseudocode for the variable bijection finding algorithm

By this algorithm we can ensure that every occurrence of a name a in the buggy code is replaced by a name b in the reference code while no other name c in the buggy code is replaced by b. This makes it safe to replace every occurrence of b in the reference solution by the student-chosen name a. When the repair algorithm for this pair is now applied again, no feedback messages for changing the variable names are generated as they now match.

However, this approach has problems when the names are important and cannot be exchanged without changes in the semantics. Listing 5.16 lists a reference solution for a function that calculates the sum of the three given attributes and returns it, while listing 5.15 uses the built-in function `len` instead of `sum`. If variable matching is applied on these two codes, the student receives no feedback as the name in the reference solution is replaced and thus no differences between the two trees exist. To solve this, names that match the name of a built-in of the Python Standard Library are not renamed.

```
def total(a, b, c, d):
    return len([a, b, c, d])
```

Listing 5.15: Buggy code for number addition

```
def total(a, b, c, d):
    return sum([a, b, c, d])
```

Listing 5.16: Reference code for circle circumference calculation

5.3.6 Finding the Minimum Repair

With the defined cost metric it is now possible to use the APTED algorithm to find the reference solution that is closest to a submitted buggy solution. To do this, the APTED algorithm is executed once for each available reference solution and calculates the edit cost between the buggy and the reference solution. Once the edit costs to all reference solutions are calculated, the minimum one is picked to calculate the necessary steps a student needs to apply to their buggy code in order to fix it, i.e., make it equivalent to the selected reference solution.

The APTED algorithm provides the necessary steps as a list of two-tuples, returned in unspecified order. Each tuple consists of a node of the buggy tree on the left side and a node of the reference tree on the right side for updates and matches, where matches mean that a node does not need to be changed and updates describe a needed inner-node change as introduced in section 5.3.4. Insertions are represented as tuples with a missing left node and a buggy node on the right side, removals vise versa. As an example, see the example in listings 5.17 and 5.18. The generated list of two-tuples of BaseNode subclasses is given in listing 5.19.

```
def function():
    var_a = 5
    print(var_a)
```

Listing 5.17: Sample code A for distance and operations calculation

```
def function():
    var_a = 7
    return var_a
```

Listing 5.18: Sample Code B for distance and operations calculation

```
[
    (Constant(5), Constant(7)),
    (None, Call(print)),
    (Return, None),
]
```

Listing 5.19: Generated tree edit operations as list of tuples for the code in listings 5.17 and 5.18

For performance reasons, this task is distributed to all available kernels using Python's *multiprocessing* standard library [36]. As every APTED calculation is fully independent from the others, no communication between the processes except from returning the calculated result is necessary. To avoid running too long, especially on a large set of student submissions, the repair algorithm can be configured to terminate the search process after a predefined timeout and select the minimum reference solution that is known so far using an environment variable. This ensures that Pyrat always returns feedback for a buggy submission for a reasonable timeout, i.e., a timeout that is longer than a single computation of the tree edit distance.

5.3.7 Linking the Trees

With the help of Python's ast module, a code can then be converted to a tree of BaseNode objects (more precisely objects of its subclasses). As an example, see the buggy and the correct code in listings 5.20 and 5.21 respectively. The goal of the codes is to calculate the surface of a circle with a given radius, while the buggy code has three errors. First, the undefined variable name *dia* is used instead of the functions argument *rad* in the calculation. Second, the power operation with the exponent 2 is missing. Instead, the student used the variable *dia* directly, i.e., it is not squared. Third, a print call is used for the variable *res* instead of a return statement.

```
def surface(rad):
    res = dia * 3.141
    print(res)
```

Listing 5.20: Buggy code for circle surface calculation

```
def surface(rad):
    res = (rad ** 2) * 3.141
    return res
```

Listing 5.21: Reference code for circle surface calculation

When parsed to a BaseNode tree, the codes result in the trees in listings 5.22 and 5.23 respectively. Each node is given an identifier, i.e., the characters A to R in front of the node names, that is not part of the tree in Pyrat but is added for reference here.

```
a: Module
  b: FunctionDef (surface)
    c: FunctionArgs
      d: FunctionArg (rad)
    e: Assign
      f: Name (res)
      g: BinOp
        j: Name (dia)
        m: BinOps (Mult)
        n: Constant (3.141)
    o: Call
      p: Name (print)
      q: CallArg
        r: Name (res)
```

Listing 5.22: BaseNode tree for the buggy code

```
A: Module
  B: FunctionDef (surface)
    C: FunctionArgs
      D: FunctionArg (rad)
    E: Assign
      F: Name (res)
      G: BinOp
        H: BinOp
          J: Name (rad)
          K: BinOps (Pow)
          L: Constant (2)
        M: BinOps (Mult)
        N: Constant (3.141)
    S: Return
      R: Name (res)
```

Listing 5.23: BaseNode tree for the reference code

```
(a, A) => match
(b, B) => match
(c, C) => match
(d, D) => match
(e, E) => match
(f, F) => match
(g, G) => match
(-, H) => insert
(j, J) => update
(-, K) => insert
(-, L) => insert
(m, M) => match
(n, N) => match
(o, -) => remove
(p, -) => remove
(q, -) => remove
(r, R) => match
(-, S) => insert
```

Listing 5.24: Operations generated for the trees in listings 5.22 and 5.23

When the APTED algorithm is executed on the trees in listings 5.22 and 5.23, a distance of 12 is calculated. In addition to the repair cost, the algorithm calculates a list of operations given in listing 5.24 that need to be performed to transform the buggy tree into the reference tree.

In these update operations, node j is mapped to node J as an update, issuing the message that the name of the variable has to be changed to *rad*. The nodes H, K, L and S in the correct tree have no counterpart in the buggy tree, thus they are listed alone in the operations list. For the feedback generation this means that those nodes have to be inserted to fix the buggy solution. In the buggy tree, node o including its two children p and q has to be removed. Only the leaf r, namely the variable *res*, has a matching node R in the reference tree, the only child of the return statement. As APTED does not remove all children of a removed node automatically, the leaf r is not removed. All other nodes are matched to their corresponding node in the reference tree without changes.

As one can see in this example, not only the values, e.g., the variable name, of a node can change. In the buggy tree, node j has node g as its parent and node m as its successor, while the corresponding node J in the reference tree has node H (a new node) as parent and node K (another new node) as its successor. Node M, linked to the successor of j in the buggy tree, is now the successor of the parent of J.

While the operations list could be used for feedback generation directly, this would have one drawback. Not every operation that is included in the operations list has to be translated to a feedback message. As an example, the feedback list from the example above contains insert operations for the buggy nodes H, K and L. Creating a feedback message for each of the insertions would instruct the student to add a binary operation with the power operator (H), the power operator (K) and the constant 2 (L). This is the case as the feedback messages include information about their children to be more precise, i.e., it would not be helpful for students if a message just advises to add a binary operation. To achieve this, the operations first have to be added to the tree so that traversal is possible. If a feedback message is generated during traversal, the message generating method can select whether further traversal is necessary or not.

For this, it is now necessary to change the buggy tree in a way that its structure is identical to the structure of the reference tree. This is done using the link property of the BaseNode class, holding a reference to the equivalent node of the other tree and the type of the link (match, update or insert). To set these links, the algorithm iterates over the list of operations generated by the configured APTED algorithm (section 2.2). Each element is a two-tuple of references, the first to the buggy node and the second to the reference node. If both values are set, the operation is either a match or an update operation, which can be determined by calculating the cost to update the buggy node to the reference node. If this cost is zero, it is a match operation, else an update operation. Both nodes are then linked and the link is marked with the according link type, i.e., match, update, insert or remove.

If the left side of the two-tuple is empty, a new node needs to be created in the buggy tree. To integrate this into the buggy tree, an empty BaseNode object, i.e., an object of the superclass, is created and linked to the equivalent reference node. If the right side of the tuple is empty, the node needs to be removed. As there is no counterpart in the reference tree, the node is added to a list of nodes that need to be removed. This list is later used for feedback generation.

```
a: Module                    - m ->  A: Module
  b: FunctionDef (surface)   - m ->    B: FunctionDef (surface)
    c: FunctionArgs          - m ->      C: FunctionArgs
      d: FunctionArg (rad)   - m ->        D: FunctionArg (rad)
    e: Assign                - m ->      E: Assign
      f: Name (res)          - m ->        F: Name (res)
      g: BinOp               - m ->        G: BinOp
        h: BaseNode          - i ->          H: BinOp
          j: Name (dia)      - u ->            J: Name (res)
          k: BaseNode        - i ->            K: BinOps (Pow)
          l: BaseNode        - i ->            L: Constant (2)
        m: BinOps (Mult)     - m ->          M: BinOps (Mult)
        n: Constant (3.141)  - m ->          N: Constant (3.141)
    s: BaseNode              - i ->      S: Return
      r: Name (res)          - m ->        R: Name (res)
```

Listing 5.25: Linked BaseNode tree for feedback generation based on the trees in listings 5.22 and 5.23. m, i and u are abbreviations for match, insert and update, respectively.

5.3.8 Generating Feedback

Based on the linked tree, Pyrat's repair algorithm can now generate feedback for the students. For this, the tree is traversed in a depth-first way, resulting in a list of feedback messages that starts with the first line of code and ends with the last one. For each update, insertion and removal, a feedback message describing the necessary change is generated. For the linked tree in listing 5.25, the feedback consists of the following messages.

1. Create a binary operation power (**).
2. Add a return statement.
3. Remove the call to the function print.

However, these feedback messages are not yet sufficient. While they describe what change has to be performed by the student, there is not enough context to know where the error is, especially in larger codes. Thus, location context is added to the messages. For updates and removals, it is possible to add the line number of the node in the buggy code, but this is not possible for insertions as they are not part of the buggy code. Instead, the location is described by the type and line number of its parent, predecessor and successor. For the three messages above, the following location context is added.

1. As part of the binary operation multiplication (*) in line 2.
   Before the constant value 3.141 in line 2.
2. As part of the function definition surface from line 1 to 3.
   After the assign statement in line 2.
3. As part of the function definition surface from line 1 to 3.
   After the assign statement in line 2.

As an additional aid the code that is to be removed and the code that needs to be inserted are included in the feedback message as well. In case of an update, both – a code to remove and a code to insert – are given to the student. However, in the Pyrat web interface (section 5.5), this information is hidden by default and can be retrieved by the student if they need further assistance. Thus, students can first try to solve the error with the feedback on their own.

5.3.9 Extendability

One advantage of the chosen design is the easy extendability in case of new Python features. In general, two changes to the Python language are possible. First, it might be the case that one of the Python language elements that have an own subclass in the ast module gets extended. In this case, the BaseNode subclass matching this ast subclass has to be extended to be aware of the changes. However, there are no Python Enhancement Proposals (PEPs) available yet that introduced such changes to an existing ast subclass [45].
Thus, the remaining option for changes to the Python language is the introduction of new subclasses to the ast module. One example for this is the introduction of the structural pattern matching, i.e., the match statement, in PEP 634 [4]. To make Pyrat's repair algorithm aware of such changes, one must add a new subclass to the BaseNode class[12]. This subclass, as any other existing subclass, receives the ast node in the constructor and stores the relevant information. Additionally, methods calculating the update cost to another node of the same type as well as methods generating the feedback texts have to be defined. To make the new node known to Pyrat, it is sufficient to add an entry to the node mapping[13]. By this, the factory method[14] can create objects for the new node when parsing the AST.

5.4 Clustering

If a student submission is correct, it does not necessarily align with an existing reference solution added by a teacher. Also, this submission represents the way of thinking of a novice solving this task, making it likely that another student uses a similar approach without getting to a correct result. Because the repair algorithm generates feedback by calculating the necessary changes to match a known reference solution, the size of the feedback is drastically reduced if the reference solution used is similar to the buggy submission (chapter 6). For this reason, Pyrat provides the ability to use correct student submissions as reference solutions in the repair process of later buggy submissions.
As described in section 5.3, the repair algorithm computes the tree edit distance of the ASTs of the buggy submission and every stored reference solution. Thus, the time needed to generate feedback increases linearly to the number of available reference solutions, including correct student submissions. In his analysis for python program equivalence, Vogt showed that depending on the exercise, up to 80 % of the submissions have "weak semantic equivalence" to another submission for this task

[12] /pyrat/src/base_repair/nodes.py
[13] `AST2NODE`, /pyrat/src/base_repair/nodes.py, lines 2301 - 2398
[14] `factory`, /pyrat/src/base_repair/nodes.py, lines 2401 - 2402

[49]. Weak semantic equivalence is thereby defined in a way that "two programs A and B are weakly semantically equivalent if they produce similar output for every possible input, assuming no undefined or generally unintended behavior" [49]. By this definition and the implementation of Pyrat's repair algorithm, this means that feedback generated for a buggy submission using two different reference solutions as a base that have weak semantic equivalence will be exactly the same.
As comparing a buggy submission against two reference solutions that are equivalent will result in equivalent feedback, it is sufficient to use one exemplary reference solution of each equivalence set. Thus, student submissions are clustered, i.e., a new correct submission is checked to see if an equivalent reference solution already exists for the problem. If this is the case, the submission is not added to the set of reference solutions as this would introduce redundancy.
The key point now is to define equivalence between two codes in a way that reduces the number of known reference solutions while offering a variety of different approaches to solve a task, reducing the amount of feedback that is generated for a buggy submission. A first way of clustering equivalent codes is used in CLARA by Gulwani et al. [10]. They define a concept of dynamic equivalent where two codes are dynamically equivalent when they have the same control flow and a total bijection between their variables, i.e., "related variables take the same values, in the same order, during the execution on the same inputs" [10]. The control flow is thereby defined as the "looping structure", where two codes have the same looping structure if they have the same loops. As an example, the programs in listings 5.26 and 5.27 would be considered dynamically equivalent as they both define one for loop and the variables take the same values. Thus, only the one submitted first would be stored as a reference solution, even if both produce a different print output. Also, tracing the assignments of all variables to check for a total bijection requires the execution of the code.
For the evaluation of program equivalence, Vogt developed an own static algorithm that clusters codes. In the implementation, the algorithm is based on first transforming the AST of a code according to certain rules to remove or align unnecessary information. Then, the ASTs can be checked for equality. A first transformer unifies all strings in Python's input prompts to *inputmessage*, making codes that prompt the user for a value in a different wording equivalent. The same is done for string arguments of the print function to unify outputs, all strings are changed to *outputmessage*. Applying these two transformers, the changes between listings 5.26 and 5.27 in lines 2 and 11 would not affect equivalence between the codes [49]. The next step is to rename the variable names. In contrast to CLARA, where the clustering algorithm tries to find a bijection between variable names, the name transformer for Vogt's analysis renames variable names after a fixed scheme in both codes. The scheme used in the provided transformer replaces every name by a generic one that respects the scope of the variable, i.e., if a code uses the same local variable name in two different functions, they would be renamed to different names by the transformer. Lastly, a transformer for assignment inlining is provided. Using this transformer, simple assignments to variables that are done in the line above the single use of these variables are removed. The value is then used instead of the variable directly [49].

```python
def function(a):
    limit = int(input("Please provide your limit value: "))
    total = 0

    for i in a:
        if i <= limit:
            print(i)
            total += i


    print("The total is", total)
    return total
```

Listing 5.26: Example code A for CLARA's equivalence

```python
def function(iterable):
    limit = int(input("limit: "))
    sumOfIterable = 0

    for x in iterable:
        if x >= limit:
            print(i)
        else:
            sumOfIterable += x

    print("The sum is", sumOfIterable)
    return sumOfIterable
```

Listing 5.27: Example code B for CLARA's equivalence

Both clustering approaches are not ideal for the clustering of correct student submissions. As the example shows, the algorithm used in CLARA will put two codes with different output in the same cluster. Also, the implementation is not static and tasks that include randomness in variables cannot be clustered correctly as it is unlikely that a variable bijection can be found. [10]
For the second algorithm, some features like the assignment inlining are not ideal as well. While two codes that are equivalent except from a variable inlining are indeed very similar and could form a cluster depending on the use case, this is not the case for program repair in Pyrat. As an example, see the correct submissions for a task in listings 5.28 and 5.29 as well as the incorrect submission in listing 5.30. If we assume that the submissions were made in this order and clustering of correct student submissions was done with the assignment inlining, submissions A and B would be clustered together. Thus, only submission A would be stored as a reference solution, assuming that there is no other equivalent code already stored. If now the incorrect code C is submitted to the task, the only real mistake made by the student is the use of the constant 3 instead of 2 in the list comprehension. Nevertheless, the student would get feedback to also create a new variable tmp and assign the return value of $outside\_function$ to it. Another feedback message would then be to use the variable tmp in the call of my_action [49]. This is an important difference to the clustering and repair algorithm used in CLARA [10].

```
def function():
    tmp = outside_function()
    res = my_action(tmp)

    return [i ** 2 for i in res]
```

Listing 5.28: Example code A for inlining, correct

```
def function():
    res = my_action(outside_function())

    return [i ** 2 for i in res]
```

Listing 5.29: Example code B for inlining, correct

```
def function():
    res = my_action(outside_function())

    return [i ** 3 for i in res]
```

Listing 5.30: Example code C for inlining, incorrect

Nevertheless, the clustering algorithm by Vogt is implemented in a modular design, thus it is possible to only use a subset of the transformations for clustering [49]. The combination of the modular design and the static execution make this cluster algorithm better suited for Pyrat than CLARA, thus it is integrated into Pyrat with some changes.

For the reasons shown beforehand, assignment inlining is not used in the clustering for Pyrat. Another change is made at the output unification. Looking at the assignments of a Python introductory lecture, they fall roughly into two categories: those that use output as part of the assignment and those that do not. For the assignments that use outputs, the correct output is usually necessary for grading, thus it should be equal for all correct submissions and correct codes will not be split into multiple clusters just because of different strings.

While submissions to assignments that do not need output usually should not contain output, this does not work in practice, as students tend to add debug outputs to their code. While unifying these outputs will help to reduce the number of clusters if two solutions both use different output formats, it will not solve the problem completely. When used as a reference solution, a code should not contain anything that is not needed to solve the task. Otherwise, students would get feedback to add certain debug outputs that are not required by the task if the reference solution used for the repair contains debug output. To solve this issue, Pyrat removes debug output while clustering instead of unifying it if it is not needed for the assignment. This is done using the same algorithm that is also used in the program repair to remove the debug output from the buggy submission (section 5.3.2). Teachers can configure the cluster settings of a task to match the need of an assignment. The same is done for the removal of top-level code and unused variables, also described in section 5.3.2.

The unification of variable names is used without changes in the clustering process. Nevertheless, the student submission is added to the set of reference solutions with its original variable names, as the generic names are in general not a good choice for variable names and would be displayed in case the submission is used as a reference solution.

While the cluster algorithm offers some options for configuration and cleaning of the submissions, it is not guaranteed that every correct student submission is a good reference solution. As an example, consider a task where a list with all keys from a given dictionary should be created where the respective dictionary value is False. To get to know the different possibilities of while and for loops, students were required to use a while loop for this task. Listing 5.31 gives an example of a real student submission that fulfills the requirements of the task and was graded with all possible points. However, the submitted solution is not of good quality. Even though if a while loop is used, it is not used in the intended way, i.e., to replace the for loop, but to replace an if statement.

```
1 results_oral_exam = {"154880": True, "379355": False, "437492": True,
2                      "212711": True, "141777": True, "340407": False}
3 failed = list()
4 for key, value in results_oral_exam.items():
5     while value == False:
6         failed.append(key)
7         break
```

Listing 5.31: Real student submission using a while loop in an unconvenient way

As bad – but correct – student submissions can occur even with the best checks, Pyrat offers multiple options for the teachers. The easiest ones are to use either only reference solutions created by the teacher, which increases the feedback size for buggy submissions, or to use all submissions, which might mean using bad submissions as reference solution.

The third option is a mix of the other two options. In the settings of a task, the teacher disables the usage of student submissions for the repair algorithm. Nevertheless, Pyrat will cluster new student submissions and store them with a *new* flag. Teachers can then review new student submissions and check if they are good enough to be used as a reference solution. If this is the case, the code's label is changed from *student* to *reference* and it will be used as a reference solution for new submissions. Otherwise, the flag is removed and the code's label remains *student*. Thus, the code is still known to Pyrat and will not pop up as a new correct submission every time a student submits an equivalent code. Lastly, teachers have the option to make some changes and then accept the code as a new reference solution.

5.5 Web Interface

Built with Next.js, the server offers both an API and a web interface by which tasks, checks and reference solutions can be added and configured. To reduce duplication of code, the web interface serves as an API-consumer, i.e., all access to the database is done by querying the API.

5.5.1 Student Interface

As the development of a web interface is not the primary objective of this master's thesis, the student interface is kept relatively straightforward. As figure 5.4 illustrates, the student is provided with the task description consisting of markdown and code cells. Below the task description, the student can work on their code for the submission. Due to its easy integration into React and thus Next.js and the rich functionality, including auto completion and syntax highlighting, Microsoft's Monaco Editor[15], primarily known from the code editor Visual Studio Code, is used.

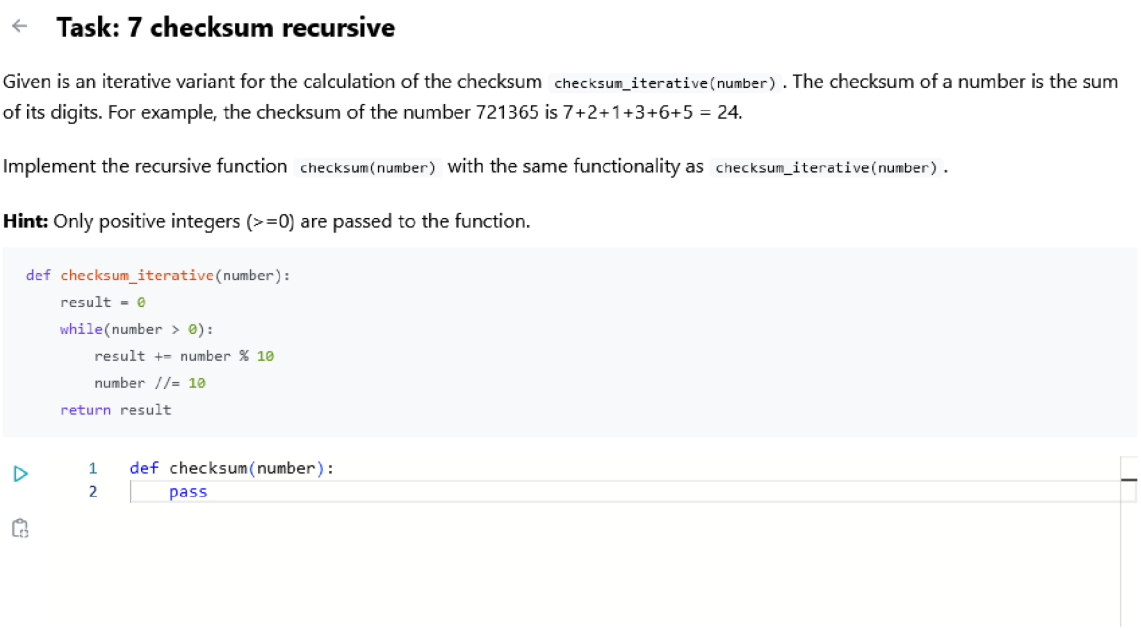


Figure 5.4: Screenshot of the student interface to work on a task

Instead of using the Pyrat interface for students, teachers are free to use the API and integrate Pyrat into any coding environment, e.g., into Jupyter Notebooks. To apply all Pyrat functions that have been configured for a task on a student submission, a single API call is sufficient. This makes integration into other environments relatively straightforward.

5.5.2 Teacher Interface

Following the workflow illustrated in figure 5.1, teachers need to start creating a Pyrat task by providing a title and description for the students. To provide the option of some styling methods while simultaneously reducing the implementation effort in the web interface, Markdown formatting can be used in the description. While a description is needed in general so that students understand how they need to solve a task, Pyrat does not force teachers to provide one. This is, for example, useful for a case where the student interface is realized by an external interface, using the API just for code repair and providing a task description on its own. As Pyrat uses the task description for the LLM-based generation of checks and reference solutions, these features are not available if no task description is provided.

[15] Monaco Editor, `https://microsoft.github.io/monaco-editor/`, last accessed 06/15/2024

For a created task teachers can then add checks and reference solutions. Besides automatic generation as described in section 5.1.1, teachers can create checks from scratch. After selecting a check type (a check from PyCheckMate or PythonTA or a unit test, see section 5.1), all the necessary arguments need to be provided. Figure 5.5 shows a screenshot of the form used to add a PyCheckMate check for the existance of a variable in the student's code. The web interface makes use of the typecheck endpoint of the check and repair container. For every argument of a check that is forwarded to the check algorithm, e.g., the variablename_to_check argument in figure 5.5, the Python type is checked. This reduces the risk of runtime exceptions when executing the checks, as those would be counted as a failed check.

Figure 5.5: Screenshot of the check adding form in the teacher interface

The teacher interface can also be used to add reference solutions to a task. When developing the code, teachers can verify them against all checks that are configured for a task. This helps ensuring that a reference solution is indeed correct and can serve as a repair base. While validating a reference solution it is also clustered against all other solutions to avoid duplicate entries.
Import and export functionality allows teachers to move tasks between different instances of Pyrat or create backups of them. A backup file is a JSON file containing all tasks with their checks and reference solutions, clustered student submissions can be included as well. Dependency relations between checks are maintained, and unique identifiers are replaced by new ones to not mix up different instances of Pyrat.

5.5.3 Application Programming Interface

While the web interface introduced in this section enables access to all parts of the system as a standalone tool, i.e., no external tools outside the scope of the Docker compose file have to be set up, it is not the only option to use Pyrat. In general, the four different scenarios listed below are possible:

1. Use Pyrat only through the web interface, including the student and the teacher interface.

2. Use the web interface to configure tasks, use the web API to check student submissions.
3. Use solely the web API, i.e., for task configuration and student submissions.
4. Use only the functionality offered by the check and repair container.

While the first option is the easiest to set up, it lacks of flexibility. Students have to use the web interface, thus extras like releasing tasks by a time condition or access management are in general not available with the current implementation of Pyrat.
Using the second option brings more flexibility as teachers can omit the student interface and implement an own solution instead, e.g., an integration into a Learning Management System (LMS). Nevertheless, the web interface can still be used to add and configure tasks instead of dealing with an API of nearly 30 endpoints (appendix C).
With the third option, this is what is done. Here, only the API is used to add and configure tasks as well as to run the check and repair algorithm on the student submissions with subsequent clustering of correct submissions. However, it is also possible to use the API only for some parts, e.g., adding reference solutions using an external source and still using the web interface to add checks to the tasks. As the API which offers the endpoints accessing the storage is realized by the web interface, the docker container for the Next.js application needs to be up and running either case.
Using the fourth option, the docker containers for storage and the web interface do not need to be running. Instead, only the check and repair container offering three endpoints for check execution, repair execution and submission clustering is required (appendix C). Thus, the user of this API has to take care of the data structure and storage themself. Additionally, it is also possible to omit the API offered by the check and repair container and import the required functionality offered by different Python classes directly.

5.6 Storage System

Persistent data storage is needed so that the different subsystems can access all the necessary data. For each task, the data listed below has to be stored for different purposes.
For Pyrat's student interface, a detailed description of the task needs to be stored. This description can then be shown to the students so they know what they have to implement. In Pyrat, a description consists of an arbitrary number of blocks, each of them either markdown formatted text or Python code.
From the teachers' perspective, every information needed for the execution of the code repair must be stored as well. This includes checks that need to be executed to check if a submission is correct or not before it can be handed to code repair. Each check can either be described by Python code that is executed (a unit test), which offers an almost unlimited range of options, or by functions from other test libraries like PythonTA [22] or PyCheckMate [3]. Every check function needs to store different information used as attributes to the Python functions implementing the checks.
Pyrat needs at least one reference solution to repair the semantically incorrect submissions. To further improve the results, it is also possible to add correct submissions from students to the pool of known correct solutions for the task (chapter 6).

As described in chapter 5, every component of the resulting correction system should run as an individual Docker container to remove obstacles in the setup process. By this constraint, the storage system should be able to run in Docker out of the box and accessing the data should be possible via network.
The most simple method in terms of setup is storing the data directly on the file system, e.g., in Comma Seperated Values (CSV) or JSON files. Nevertheless, providing access via network is more complicated here as a custom server script to read and write the files, organize them and deliver the content via network access is needed. Also, scalability issues can arise and have to be solved by custom code.
To avoid the manual structuring of the data on the file system, a database management system can be used. Due to their prevalence, MariaDB and MongoDB are considered an option. While MariaDB as a relational SQL-based database is good for data that is always structured in the same way, MongoDB is more flexible. As every check has different arguments that need to be defined, it is easier to store these in MongoDB than in MariaDB. Also, n:n-relations defining dependencies between checks can easier be stored in MongoDB.
As MongoDB provides an official docker image[16], setup is possible across many operating systems and environments.
A first collection *tasks* is used to store the task definition, including a title, a description and possible other future values.
A second collection *checks* stores the single checks configured for each task. This includes the id of the execution function, a title, feedback for the students, arguments for the execution function and a list of other checks this one depends on. Even if it was be possible to store the checks as a subdocument in the tasks collection, moving them into an own collection improves the performance when editing them.
With a third collection *solutions* the database is able to store the reference files created by the teacher as well as the correct student submissions.

[16] mongo - Official Image | Docker Hub, `https://hub.docker.com/_/mongo`, last accessed 07/25/2024

Chapter 6 Evaluation

The evaluation of Pyrat is split into two parts. The first part contains a questionnaire with Python experts, reviewing the feedback generated by Pyrat for eight real student submissions to four different tasks. The second part of the evaluation analyzes the repair improvements described in sections 5.3.2 and 5.3.5.

6.1 Building the Questionnaire

The goal of the questionnaire is to determine the quality of the feedback that Pyrat generates and to detect possible improvements for the future. While the web interface is used in the evaluation process, it is not part of the questionnaire to evaluate the usability of the interface.

For practical reasons, i.e., no available Python course for novices at the time of the evaluation, a questionnaire designed for students working with Pyrat as part of their exercises was not possbile. Thus, we decided to design the questionnaire for Python experts and educators. To evaluate the feedback generated for realistic buggy student submissions, we reused real student submissions from an introductory Python course, offered in winter semester 2021/2022.

The questionnaire was administered online using SoSci Survey[1], the Pyrat tool was also offered online, i.e., no setup was necessary for the participants. A Zoom[2] meeting was used to provide assistance with problems and questions.

6.1.1 Selecting the Tasks

From all the tasks and submissions available from the lecture, we selected the following four tasks with eight buggy submissions for the evaluation. By selecting these tasks, we can evaluate tasks that cannot be repaired with the tools listed as Related Work in chapter 3. The Random List task cannot be checked by the simple input-output tests used in Refactory due to randomness in the output. Also, the object orientation required in the Vehicle Class cannot be repaired by CLARA or Refactory. [10, 14].

[1] SoSci Survey - professionelle Onlinebefragung made in Germany, `https://www.soscisurvey.de/`, last accessed 08/04/2024

[2] One platform to connect | Zoom, `https://zoom.us/`, last accessed 08/04/2024

Average Price Define a function `average_price_petrol`. Your function should accept an arbitrary number of keyword parameters and nothing else. You can assume that every value passed as a keyword parameter is either of type `int` or `float`. Your function may behave arbitrary if something else is passed to your function, including a positional parameter of any type.
Your function should calculate the mean of the given values and return it, rounded to three decimal digits. You may use the built-in `round` function, but you may not import any modules.

Car Dictionary Create a dictionary, named `cars`, that includes cars from different car brands as a set respectively.

- VW: Golf, Polo, up!
- Seat: Leon, Ibiza
- Daihatsu: Sirion
- Audi: A4, Q3, R7, Q2

Random List Create the function `create_random_list(...)`, which has three parameters: `size`, `min_value`, `max_value`. This function shall return a randomly generated list of integers of length size where the list values are between `min_value` and `max_value`. You can assume that only correct data is passed to the function.
For example, `create_random_list(5, -85, 53)` should return a list like `[-83, 42, 53, -15, 0]`.

Vehicle Class A classic example of object-oriented programming are vehicles. Write the class Vehicle with the following specifications:

- Attributes: `colour`, `construction_year`, `mileage` (the init function should get them in this order)
- Example for string representation with `colour = white`, `construction_year = 2000` and `mileage = 123456`: *The vehicle is white, was manufactured in the year 2000 and has run 123456 kilometers so far.*
- A method `drive(...)` which receives a distance in kilometers and adjusts the vehicle's mileage accordingly. In the case of negative distances, the vehicle's mileage is of course not changed.

The student submissions selected for these tasks are given in appendix B.

6.1.2 Repair Options

The feedback differs depending on which repair improvement options described in sections 5.3.2 and 5.3.5 are used for a task. Due to the time needed for a teacher to understand and evaluate a generated feedback, it is not possible to generate feedback for every combination of the improvements and use it in the evaluation process. For consistency reasons we thus decide to use all of the implemented improvements, as section 6.3 shows a general reduction in the number of error messages.

However, using known correct student submissions might influence the generated feedback in a negative way, if the used student submission is of bad quality. To evaluate this, we show two feedback sets for the evaluation. Feedback set A is generated only with teacher-created reference solutions while feedback set B uses all available solutions.

6.1.3 Evaluation Questions

The questionnaire is divided into eight rounds with the same questions, each of them handling one buggy student submission. Figure 6.1 presents a screenshot of the first page of each round, exemplary for a student submission to the car dictionary task.
The objective of the initial question is to familiarize the participant with the task and the buggy submission. Furthermore, the answer can be utilized as a reference when analyzing the participant's feedback regarding Pyrat's proposed repair. The unit test result displayed below the submission can be employed by the participant to identify the error in the code more efficiently.

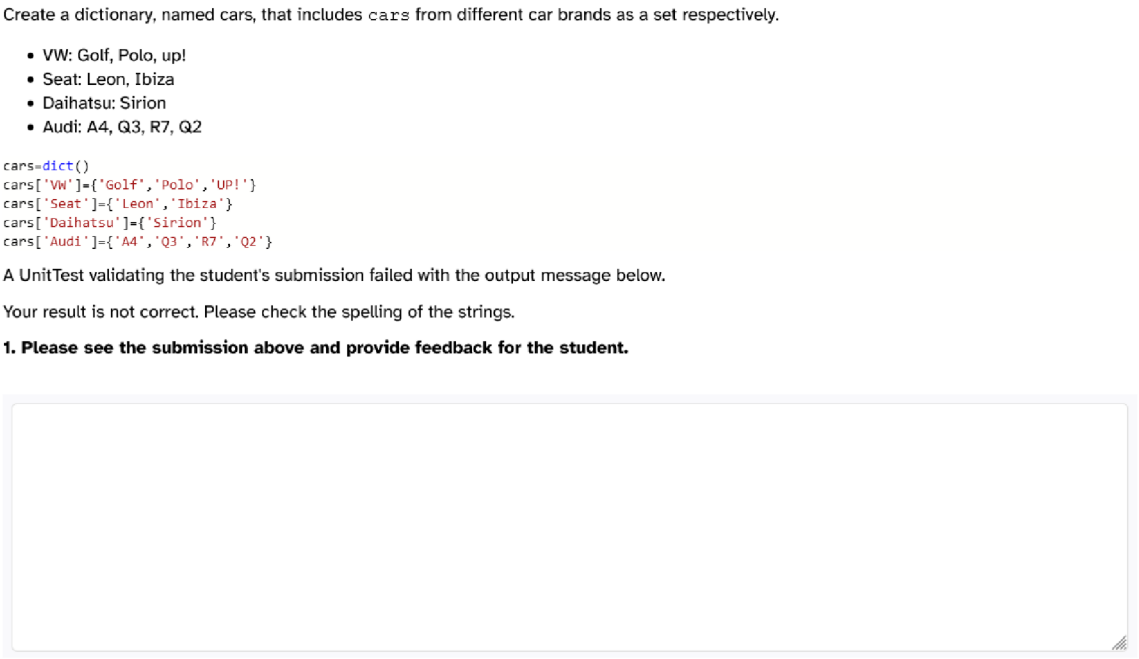
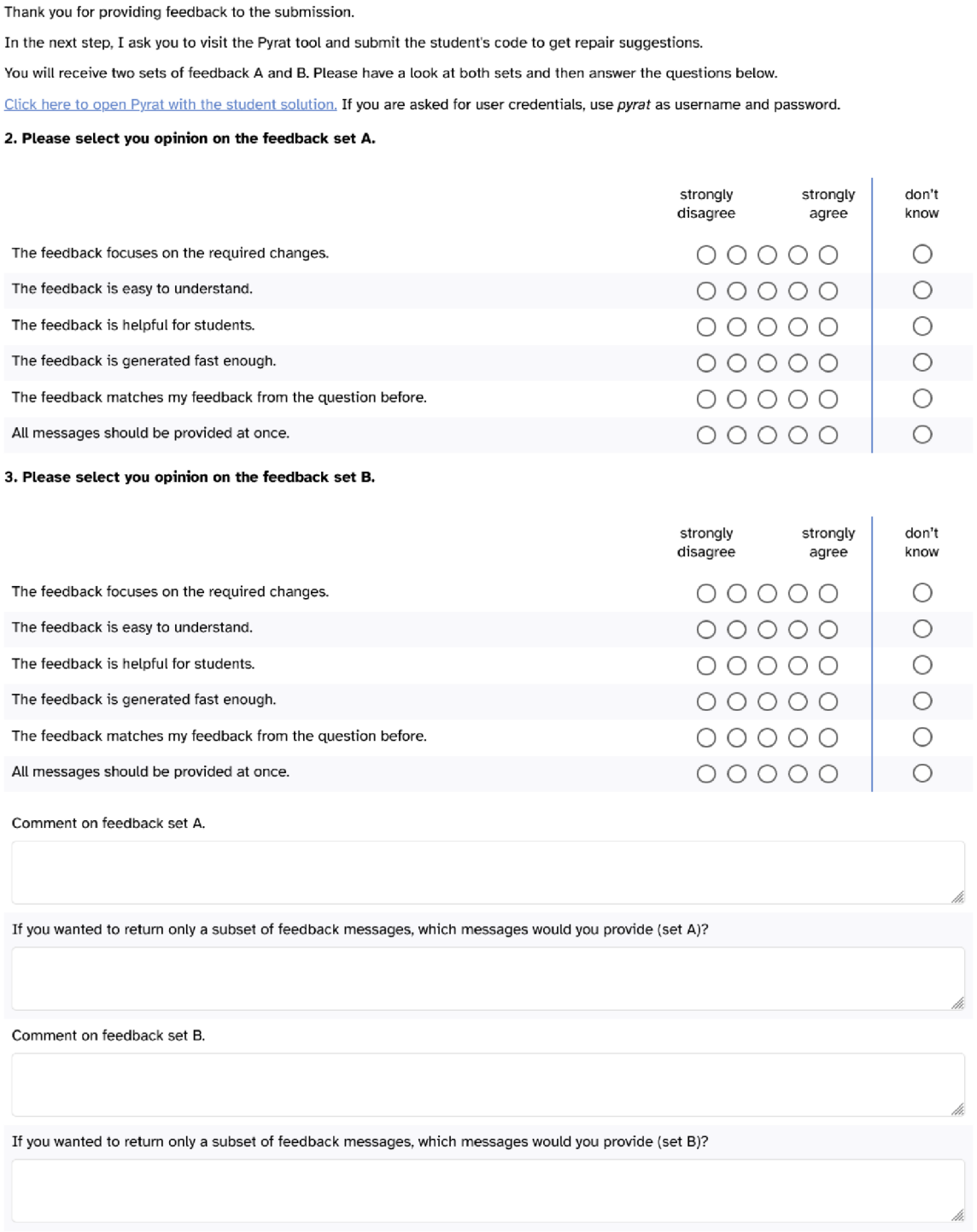


Figure 6.1: First page from the evaluation of the car dictionary task, familiarizing the participant with the task and the submission

On the next page (figure 6.2), the participant is redirected to the student interface of Pyrat to submit the buggy code and analyze the generated repair suggestion. A five-point Likert-scale is used to obtain an overall assessment on the repair suggestion.
Due to the conception of the repair algorithm used in Pyrat, the repair suggestion is a step-by-step tutorial on how to align the buggy solution with a correct reference solution. While this leads to a correct result, there might be an easier way to repair the code, especially if no similar reference solution is available. Thus, the first three subquestions verify that the repair suggestion provides the necessary information without additional, possibly distracting, steps.



Thank you for providing feedback to the submission.

In the next step, I ask you to visit the Pyrat tool and submit the student's code to get repair suggestions.

You will receive two sets of feedback A and B. Please have a look at both sets and then answer the questions below.

Click here to open Pyrat with the student solution. If you are asked for user credentials, use *pyrat* as username and password.

2. Please select you opinion on the feedback set A.

	strongly disagree			strongly agree	don't know
The feedback focuses on the required changes.	○	○	○	○	○
The feedback is easy to understand.	○	○	○	○	○
The feedback is helpful for students.	○	○	○	○	○
The feedback is generated fast enough.	○	○	○	○	○
The feedback matches my feedback from the question before.	○	○	○	○	○
All messages should be provided at once.	○	○	○	○	○

3. Please select you opinion on the feedback set B.

	strongly disagree			strongly agree	don't know
The feedback focuses on the required changes.	○	○	○	○	○
The feedback is easy to understand.	○	○	○	○	○
The feedback is helpful for students.	○	○	○	○	○
The feedback is generated fast enough.	○	○	○	○	○
The feedback matches my feedback from the question before.	○	○	○	○	○
All messages should be provided at once.	○	○	○	○	○

Comment on feedback set A.

If you wanted to return only a subset of feedback messages, which messages would you provide (set A)?

Comment on feedback set B.

If you wanted to return only a subset of feedback messages, which messages would you provide (set B)?

Figure 6.2: Second page from the evaluation of the car dictionary task, getting feedback from the participant regarding the proposed repair

While the code repair is always performed in no more than 20 seconds for the given tasks, the fourth subquestion aims to get an estimate of whether further speed improvements are needed for the repair algorithm.

Under the assumption that a skilled teacher as participant in this questionnaire provides feedback of good quality, a similarity between the repair suggestion and the participant's feedback from before shows that the repair suggestion is of good

quality as well. The fifth subquestion verifies the similarity between the participant's feedback and the repair suggestion.
As a five-point Likert-scale is not sufficient for detailed feedback, an additional free-text field can be used to comment the feedback sets. Another free-text field can be used to specify which feedback messages should be returned, if this was selected in the according subquestion before.
In the end of the questionnaire, the participants are asked to rate their Python and their teaching skills between *none existant* and *excellent*.

6.2 Evaluating the Results

A total of seven participants completed the questionnaire, taking an average of 64 minutes. As visualized in figure 6.3, the participants evaluate their Python skills between average and excellent. However, teaching experience variates between none existing and excellent, representing the range of participants from two master students to a postdoctoral researcher.
Figure 6.4 shows the results of the Likert-scale questions, ranging from 1 *strongly disagree* to 5 *strongly agree*. In every subgraphic, the result to one of the six questions is given for all of the tasks and submissions. The stacked colored bars represent the absolute number of votes for the five options of the Likert-scale. Additionally, the blue bars represent the mean value. Every data set is labeled with the task's name, e.g., Average Price, and the feedback set, i.e., A or B for the repair without or with the correct student submissions, respectively. If neither A nor B is given, the repair suggestions are the same for both cases, so only one has been provided for the participants. The numbers are only used to refer to the different submissions for a task.
For the further evaluation of the questionnaire we categorize the solutions into three categories. The first category contains submissions where both repair suggestions receive mainly disagreement in the Likert-questions, i.e., the submissions Average Price 2 and Vehicle Class 2. The second category contains submissions where the acceptance of repair suggestions B could be significantly improved compared to those for repair suggestions A, i.e., the submissions Average Price 1, Car Dictionary 1, Car Dictionary 3 and Random List 1. The last category contains submissions where only one repair suggestion was used and received mainly agreement in the Likert-questions, i.e., the submissions Car Dictionary 2 and Vehicle Class 1. In both cases, we only used one repair suggestion in the evaluation because the provided reference solution is also the minimal repair when the correct student solutions are also available, i.e., both repair suggestions are built on the same reference and thus equal.

6.2.1 Negative Feedback for Suggestions A and B

As Figure 6.4 shows, the acceptance of the feedback for the submission Average Price 2 consisting of 13 or 8 messages (feedback set A and B, respectively) is low. An often mentioned cause for this rating is that the feedback completely rewrites the code and "enforces a specific way of computing the output and iterating the input" [P5]. This is the case because the student approach completely differs from the approach

used in the reference solution and also all correct student submissions. However, the feedback says that "each feedback fragment itself is easy to understand" [P1]. In total, all participants agree that only the message for the return statement should be returned.
Another submission with negative feedback for both set A and B is Vehicle Class 2. P2 states that "the addition of the if-statement in the drive function is realized by suggestion [sic!] the complete removal of the function and the insertion of the fixed version. This gets the student to the correct solution, but is not really helpful." As the tree edit distance always computes the minimal edit to repair a buggy solution, this indicates an error in the applied cost metric that prefers removing a small function instead of adding an if condition into it. Another critic is that the repair suggestions force the student to change the `__str__` method of the class, even if the output is correct.

6.2.2 Negative Feedback for Suggestion A, Positive for B

In contrast to the previous two submissions, the feedback could be improved by using correct student submissions for the submissions in this section. The repair suggestion set A for the submission Average Price 1 is described as "overly complicated" [P5] with some of the messages being "hard to understand" [P6]. While the only reference solution available for set A iterates directly over the values of the dictionary, the buggy student submission iterates over the keys of the dictionary and uses subscription to retrieve the values. Thus, two messages advise the student to change this behaviour, one for the change in the loop head and one for the occurrence in the loop body. However, the two messages are not listed together, but as the first and the fourth messages. Repair suggestion set B, on the other hand, is computed with a reference solution that also iterates over the dictionary keys, reducing the number of messages to the necessary one telling the student to change the parameter of the `round` function from 4 to 3. In the questionnaire, this is rated as "easy to understand and helpful to fix the problem" [P4]. However, P6 states that this suggestion "solves [the] obvious mistake" while the "solutions is still not perfect". Combined with the subset of messages P6 wants to return from repair suggestion set A and the comment that "it would be helpful to differentiate between errors and improvements" [P4], this indicates that Pyrat needs to implement functionality to present the feedback in a sensible sequence and extended with context information, i.e., whether it is a mandatory fix or just an improvement.
Similar holds for the submissions Car Dictionary 1 and Car Dictionary 3, where repair suggestion A rewrites the entire code to a single-step dictionary definition. However, the error in the submissions is the spelling of the string "up!" and the sets all having just one element, a comma-separated string of models, respectively. This is correctly provided as feedback in the repair suggestion set B.
The submission for Random List 1 misses a return statement, thus the error is very similar to the one in Average Price 2. However, using student submissions as references now improves the repair suggestion B in a way that just a single suggestion (adding a return statement) is provided to the student. As figure 6.4 shows, all participants selected "strongly agree" on all Likert-questions that evaluate the quality of the feedback. The only exception is P5 with a Likert score of 3 for the question

in figure 6.4(e). This indicates that the feedback provided by P5 differs from the feedback generated in repair suggestion B. For comparison, the feedback for the student provided is "Please retunr [sic!] your created list instead of printing it (which is not the same!). Use 'return list' instad [sic!] your last line." [P5]. While it can be a helpful information to the student that there is a difference between printing and returning a result, one can discuss if it is really necessary to remove the print statement as the task description does not make any specifications regarding debug outputs, i.e., it is also okay to have both a print statement and a return statement.

6.2.3 Positive Feedback for the Only Suggestion

Figure 6.4 shows overall acceptance for the submissions Car Dictionary 2 and Vehicle Class 1. However, while the participants state that the repair suggestions for Car Dictionary 2 focus on the required changes (figure 6.4(a)), the acceptance for the question if the feedback is easy to understand is lower (figure 6.4(b)). This difference is supported by the comments, e.g., it is criticized that "there are three suggestions for the same change (tuple to set) but different lines" [P3]. The same is added by P4 ("It could be shortened as it is the same for every line but I don't think that is necessary and it is helpful as it is.") and P6 ("Just make all of them as one message.") For the submission Vehicle Class 1, on the other hand, no further critic is given. The repair suggestion is described as "helpful and understandable" [P4], P5 states that "The feedback is simple to understand and correct."

6.2.4 General Results

Interestingly, the answer to the question "The feedback is generated fast enough" (figure 6.4(d)) differs between feedback sets A and B for three of the six tasks with two feedback sets. However, the customized version of Pyrat for the questionnaire first generates both feedback sets and then sends them to the web interface, where both are displayed simultaneously for technical reasons. Thus, the perception of whether the time needed is short enough or not seems to depend on the feedback generated. Participants tolerate a higher repair time for the feedback that generally is rated better and consists of fewer feedback messages, as shown in figure 6.5. On the technical side, this feedback does indeed take more time to generate, because the tree-edit distance has to be computed against more reference solutions compared to just using the teacher-authored reference solutions.

6.3 Evaluation of Prefiltering Functions

Besides the questionnaire we perform an evaluation of the repair improvements. For this evaluation, we select six tasks from an introductory Python lecture, i.e., the four tasks already known from section 6.1 and the two tasks listed below. For these tasks, 1,358 student submissions are available in total, 407 or 29.97 % of them failing at least one of the configured unit tests, i.e., containing errors. Table 6.1 provides detailed information about the number of submissions as well as the error and repair rate per task. The difference between error and repair rate is caused by tasks that fail one of the configured formal checks, e.g., do not compile or do not define the function required by the task.

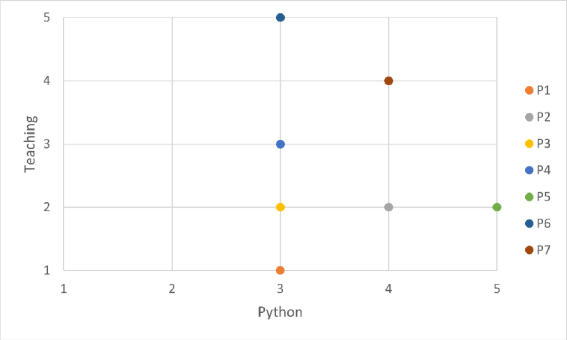


Figure 6.3: Self assessment of the Python skills and teaching experience. Ranges from 1 (none existent) to 5 (excellent).

Exercise Set Assume you have a list `expectedValues`. During an experiment, data is stored in a list `collectedData`. For the further process, it is relevant which datapoints match. Therefore, create two sets by using appropriate methods:

- `matches` contains all datapoints that are common for `expectedValues` and `collectedData`
- `nonMatches` contains all data that was expected but not collected

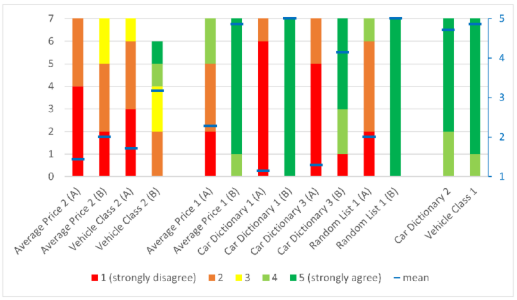
Recursive Checksum Given is an iterative variant for the calculation of the checksum `checksum_iterative(number)`. The checksum of a number is the sum of its digits. For example, the checksum of the number 721365 is 7+2+1+3+6+5 = 24.
Implement the recursive function `checksum(number)` with the same functionality as `checksum_iterative(number)`.
Hint: Only positive integers (>=0) are passed to the function.

```
1 def checksum_iterative(number):
2     result = 0
3     while(number > 0):
4         result += number % 10
5         number //= 10
6     return result
```

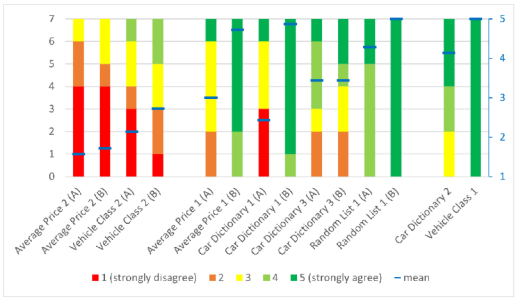
Task	Total sub.	Error rate	Repair rate	Corr. clusters
Exercise Set	232	24.14 %	23.71 %	66
Car Dictionary	233	26.18 %	25.32 %	51
Average Price	232	33.62 %	28.02 %	113
Random List	228	13.16 %	13.16 %	70
Recursive Checksum	219	13.24 %	5.02 %	85
Vehicle Class	214	71.50 %	71.03 %	40

Table 6.1: Number of available submissions for the selected tasks with their error and repair rate

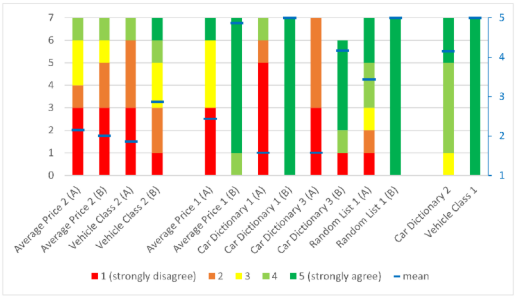
As noted above, this evaluation focuses on evaluating the performance of the repair improvements introduced in sections 5.3.2 and 5.3.5, i.e., removing debug output, removing top level code, removing unused variables and variable name matching.



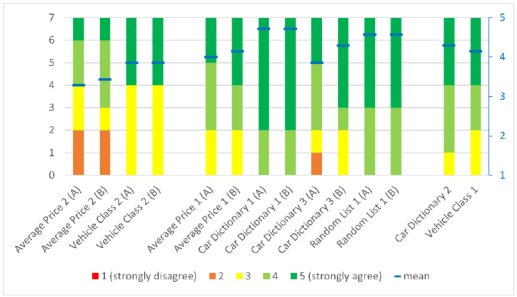
(a) The feedback focuses on the required changes.



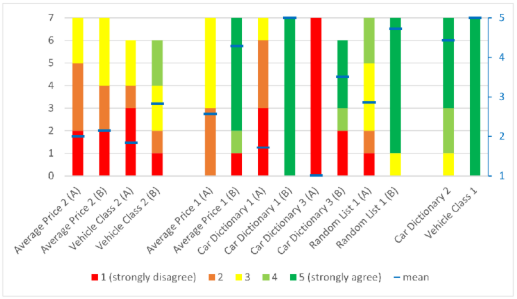
(b) The feedback is easy to understand.



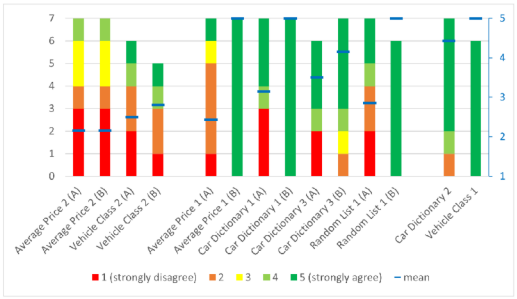
(c) The feedback is helpful for students.



(d) The feedback is generated fast enough.



(e) The feedback matches my feedback from the question before.



(f) All messages should be provided at once.

Figure 6.4: Evaluation results of the Likert-scale questions. Missing values are caused by the "don't know" option.

All of these improvements can be tested with or without the use of correct student submissions as references in the repair algorithm. This results in five different characteristics and thus 32 different combinations that can be compared with each other in this part of the evaluation. However, not every combination needs to be evaluated because some of the features do not affect each other. As an example, removing debug output works whether or not variables are matched before. As a compromise, we run all of the 407 buggy submissions on the six repair configurations listed in table 6.2.

Batch ID	Student sub.	Debug	Top level	Unused var.	Match
A	no	no	no	no	no
B	yes	no	no	no	no
C	yes	yes	no	no	no
D	yes	no	yes[3]	no	no
E	yes	no	no	yes[3]	no
F	yes	no	no	no	yes

Table 6.2: Configurations used for the evaluation of the prefiltering options

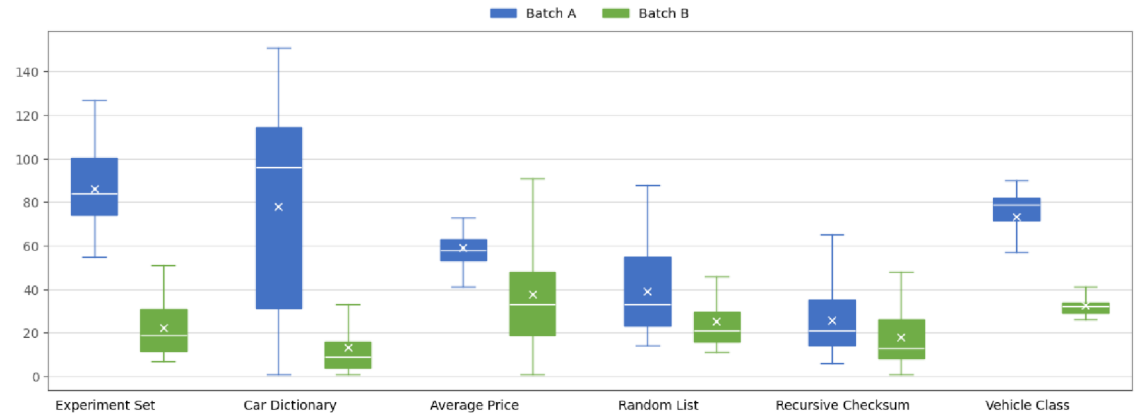
6.3.1 Usage of Correct Student Submissions

The boxplot in figure 6.5 shows the comparison between the batches A and B, i.e., a wider view on the tasks used for the questionnaire as well. As one can see, the cost of the suggested repairs (using the metric presented in section 5.3) decreases as well when using the correct student submissions. In detail, the cost is reduced from an average of 69.3 in batch A to 28.0 in batch B, reducing the cost by 60 %. However, the number of generated repair messages is only reduced by 23 %, i.e., from 8.9 to 6.8. This results in an average cost per repair message of 7.8 and 4.1 in batch A and B, respectively, indicating that each feedback message is easier to understand as it describes a smaller change.
As an example, see the feedback for the code 3 to the task Car Dictionary in appendix B. The student's approach is to first create an empty dictionary and then add the four needed entries each by an own subscript. The sets are thereby created using a list converted to a set with Python's built-in `set` function. However, the student created lists with a single string each containing the car models separated by comma instead of an own string for each model.
Without using the known student submissions as references as well, feedback advises the student to remove the dictionary and the assignments and instead create a new dictionary containing all the data at once. This is caused by the fact that the solely known reference solution defines the dictionary in one step and thus is not usable for the approach of the student that is in general correct as well. Including the correct student submissions as well, on the other hand, reduces the repair cost while simultaneously increasing the number of repair messages. The student is now adviced to change each of the four string constants to only the first model and then add new string constants for the other models to the list. While this now results in nine feedback messages, they are easy to understand as they are all similar. Nevertheless, clustering similar messages can be useful to reduce the number here and is discussed in chapter 7.

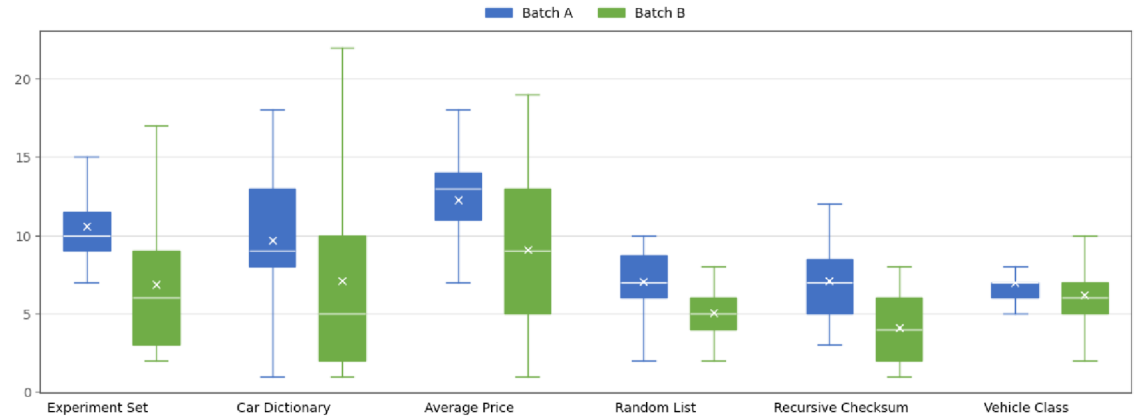
6.3.2 Removing Debug Output

While using correct student solutions reduces the feedback size by finding a more similar reference solution for a buggy code, this and the following improvements reduce the number of feedback messages by prefiltering and transforming the buggy

[3] The tasks Exercise Set and Car Dictionary are not included here as they do not use functions, i.e., they consist only of top level code and/or unused variables.



(a) Repair cost



(b) Number of repair messages

Figure 6.5: Boxplots representing the repair cost and number of repair messages comparing batches A and B

.

code for certain aspects. The first improvement removes debug output from the buggy submission before the repair algorithm is applied (section 5.3.2). This prevents feedback messages that ask students to remove print statements from the code that are not actually incorrect. As figure 6.6 (batches B and C) presents, this reduces the repair cost and the number of feedback messages slightly. The average repair cost and the average number of feedback messages over all submissions is thereby reduced by 16 % (28.0 vs 23.3 or 6.8 vs 5.7, respectively).

6.3.3 Removing Top Level Code

As the tasks Exercise Set and Car Dictionary do not use functions but instead just create variables, removing top level code would remove the whole submission here. Thus, the evaluation for this repair improvement is only performed with the remaining four tasks that have 290 buggy codes in total. As shown in figure 6.6 (batches B and D), removing top level code reduces the repair cost and number of repair messages for all of the four tasks. In total, the average repair cost and the average number

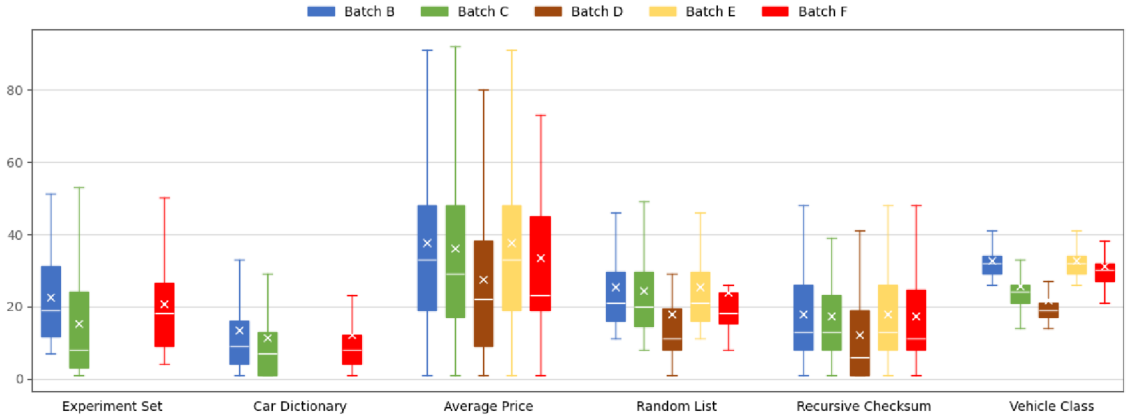
of feedback messages over all submissions of the four remaining tasks is reduced by 31.4 % or 27.5 % (32.5 vs 22.3 or 6.7 vs 4.9, respectively).

6.3.4 Removing Unused Variables

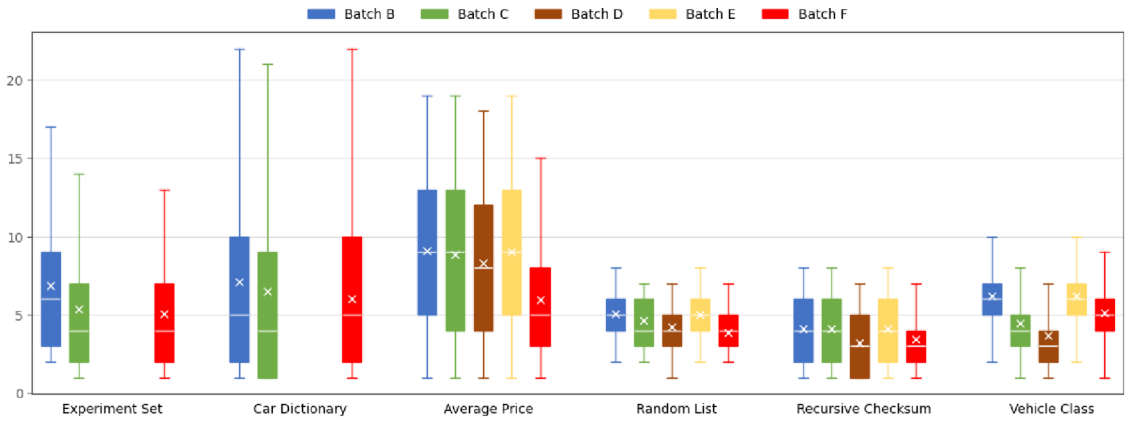
As for the evaluation of removing top level code, the tasks Exercise Set and Car Dictionary cannot be used as their goal is to define a variable that is not used. Results in figure 6.6 (batches B and E) show that the average repair cost over all remaining 290 buggy codes can only be reduced by 0.05 %, resulting in a reduction of the average number of feedback messages by 0.23 %. However, the ability to detect and remove unused variables is still useful for Pyrat. As described for the implementation of the clustering in section 5.4, variable names are unified to a predefined scheme to cluster codes that are equivalent except from variable naming. Since the unification algorithm is based on numbering variables in the order of their first occurrence, the existence of unused variables will disturb the unification. An alternative to using this option in the repair process is to enable the PythonTA check for unused variables as a warning, issuing a feedback message also for correct submissions with unused variables instead of forcing students to remove the variable in the repair process.

6.3.5 Variable Name Matching

The last one of the available improvements is the matching of variable names to reduce messages that require the student to unnecessarily change the name of a variable. It can be applied on all six selected tasks as this improvement does not remove any code. Figure 6.6 (batches B and F) shows a general reduction of the repair cost and the number of repair messages. In total, the average repair cost and the average number of feedback messages over all submissions is reduced by 6.7 % or 23.8 % (32.5 vs 30.3 or 6.7 vs 5.1, respectively). As variable name changes are relatively cheap operations per feedback message, the reduction of the number of messages compared to the reduction of cost is higher for this improvement than for the others before.



(a) Repair cost



(b) Number of repair messages

Figure 6.6: Boxplots representing the repair cost and number of repair messages comparing batches B to F

.

Chapter 7 Future Work

Based on the evaluation in chapter 6, we develop a number of aspects how the functionality of Pyrat should be extended in the future. However, due to time limitations, is is not possible to implement the extensions in this master's thesis.

Message Ordering In the current version of Pyrat, insert and update instructions are listed in the order of their occurrence in the code, i.e., starting with messages for the first line. However, delete instructions are added after all other messages as they are not part of the tree that is traversed for message generation. This can be confusing, especially if a delete and insert refer to the same location.

Even when related insert and delete operations are displayed in a contiguous order, the first-to-last line order used must not be ideal. As an example, P3 suggests to "consider reordering the feedback messages: Show the mandatory ones (for the correct solution) at first and group the 'performance' messages (or clean code) at the end of the list". Thus, further research is necessary on how the order of the feedback improves the learning experience of the students.

Release Feedback Step by Step This is especially important for the suggested option to not reveal all feedback messages at once, as requested by P1. One possibility would be to start with general feedback providing an abstract description of what needs to be changed. In case this does not help the student, more detailed feedback can be revealed. This is a continuation of the already used approach of initially hiding the concrete source code of the feedback messages and only showing it when the student requests it. A positive side effect of this is a better feedback for empty solutions. Using the repair algorithm in its current state, Pyrat would generate a set of insert messages creating the shortest reference solution. By revealing only a subset of messages, students could use them as a starting base and continue to work on the code on their own instead of copying the reference solution step-by-step.

Keep Repair Context Over Multiple Executions To provide consistent feedback, Pyrat can be extended to store session context when used by a student. Thus, the same reference solution can be used as source of feedback generation over and over if a student makes changes on their code, but the code still contains errors. This avoids contradicting messages between multiple executions with changes on the code.

Feedback Message Clustering Next, multiple feedback messages should be combined together if they are very similar, e.g., if multiple messages advise the student to change a tuple into a set (section 6.2). However, it is possible that this will be solved by revealing the feedback messages step by step, as students may find that the error occurs in several places on their own. Further research is necessary for this.

Larger Code Snippets for Easier Repair In general, Pyrat traverses the Abstract Syntax Tree (AST) tree up to the first node that contains location information to provide this information and the code change to the student. However, this can result in very small code snippets, e.g., if a constant has to be replaced by another constant, only those two constants are given, making the additional information redundant to the information already contained in the text. Thus, it is an alternative to always traverse the AST tree up to the full line, i.e., display the line of code that is to be removed or inserted.

Improved Variable Matching While the variable matching shows a positive affect on the repair size and the number of repair messages especially for function- or class-based tasks, it is not perfect. As a bijection has to be found to match two variables, there will be situations where feedback messages are generated that require the student to rename a variable while it is not necessary. This variant is chosen for the current implementation of Pyrat as it guarantees correctness of the refactored reference solution. However, a combination of an unsafer variable matching algorithm and testing of the refactored reference solution might improve the repair cost further as more variable names can be matched. Additionally, it is necessary to enable the teacher to further restrict variable matching, i.e., define names that are not allowed to change in the reference solution for external reasons.

Introduce Reordering to the Tree Edit Distance Algorithm The tree edit distance algorithm used in Pyrat applies the three operations insert, remove and update to describe the changes between two ASTs. However, this results in confusing feedback messages in the case of wrong sequence, e.g., of function arguments. With the current configuration, Pyrat will correct the argument sequence d, b, c, a to a, b, c, d by two repair messages: one removing the argument a after the argument c, and another one to create a missing argument a before b. While this problem can already be reduced by clustering the two repair messages together or listing them contiguously (see the possible improvements listed above), it is also an option to add a move operation to the tree edit distance algorithm.

Another use case for a move operation is for code segments where the order of elements in the AST is not relevant. As an example, this is the case for a `ast.BinOp` operation with the addition operator. Due to the commutative law, the order of the left and right expression is not revelant for the result here. Pyrat should be aware of this to prevent unnecessary repair messages. The same holds for method definitions in a class, as their ordering does not affect the execution as long as no method is overwritten, i.e., two methods are defined with the same name.

Multiple Reference Solutions for Single Repair In the context of classes or more complex tasks with multiple functions, another improvement is the use of multiple reference solutions for different parts of the submission. Thus, two buggy methods a and b could use a different reference solutions if each of the reference solution is only similar to the buggy submission in one of the methods. However, this needs further checks before returning the feedback as it might generate erroneous feedback, e.g., if b is an auxiliary function and not covered by unit tests directly. If the reference version of a is not compatible to the reference version of b, the overall feedback will not result in a working code.

Restrict Repair to Buggy Subparts of the Submission Another option can be to enable the repair algorithm on subparts of the submission, e.g., a single method of a defined class, only if a specific unit test for this method failed. Thus, Pyrat could prevent feedback that fixes a correct method of a class just because the nearest reference solution used for another method choses another approach for this method. This is, as an example, the case for the submissions to the Average Price task used in the evaluation.

Final Remarks In conclusion, Pyrat is a tool that generates useful feedback for small errors. However, for the correction of complex errors, further research and testing with a larger number of students is necessary. Due to the modular design, the availability of an Application Programming Interface (API) and the full Python support of the submitted version, integration into existing courses for testing and evaluation purposes is possible.

Appendix A Bibliography

[1] M. Rifky I. Bariansyah, Satrio Adi Rukmono, and Riza Satria Perdana. Semantic Approach for Increasing Test Case Coverage in Automated Grading of Programming Exercise. In *2021 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6, Bandung, Indonesia, November 2021. IEEE. URL: https://ieeexplore.ieee.org/document/9648439/, doi:10.1109/ICoDSE53690.2021.9648439.

[2] Sahil Bhatia and Rishabh Singh. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks, March 2016. arXiv:1603.06129 [cs]. URL: http://arxiv.org/abs/1603.06129.

[3] Annabell Brocker and Ulrik Schroeder. pycheckmate – Addressing Challenges in Automatic Code Evaluation and Feedback Generation for Python Novices. page 10.18420/abp2023. Gesellschaft für Informatik e.V., 2023. URL: https://dl.gi.de/handle/20.500.12116/42565.

[4] Brandt Bucher and Guido van Rossum. PEP 634 – Structural Pattern Matching: Specification | peps.python.org, July 2024. last accessed 07/21/2024. URL: https://peps.python.org/pep-0634/.

[5] Cheat Sheets Series Team. Docker Security - OWASP Cheat Sheet Series, July 2024. last accessed 07/23/2024. URL: https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html.

[6] Paul Denny, Viraj Kumar, and Nasser Giacaman. Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language, October 2022. arXiv:2210.15157 [cs]. URL: http://arxiv.org/abs/2210.15157.

[7] Stephen H Edwards and Manuel A Pérez-Quiñones. Web-CAT: Automatically Grading Programming Assignments. July 2008.

[8] Onyeka Ezenwoye. What Language? - The Choice of an Introductory Programming Language. In *2018 IEEE Frontiers in Education Conference (FIE)*, pages 1–8, October 2018. ISSN: 2377-634X. URL: https://ieeexplore.ieee.org/abstract/document/8658592, doi:10.1109/FIE.2018.8658592.

[9] Michael H. Goldwasser and David Letscher. Teaching an object-oriented CS1 -: with Python. *ACM SIGCSE Bulletin*, 40(3):42–46, August 2008. URL: https://dl.acm.org/doi/10.1145/1597849.1384285, doi:10.1145/1597849.1384285.

[10] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 465–480, Philadelphia PA USA, June 2018. ACM. URL: https://dl.acm.org/doi/10.1145/3192366.3192387, doi:10.1145/3192366.3192387.

[11] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D. Nguyen. Providing Meaningful Feedback for Autograding of Programming Assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 278–283, Baltimore Maryland USA, February 2018. ACM. URL: https://dl.acm.org/doi/10.1145/3159450.3159502, doi:10.1145/3159450.3159502.

[12] Jack Hollingsworth. Automatic graders for programming classes. *Communications of the ACM*, 3(10):528–529, October 1960. URL: https://dl.acm.org/doi/10.1145/367415.367422, doi:10.1145/367415.367422.

[13] Krystal Hu. ChatGPT sets record for fastest-growing user base - analyst note. *Reuters*, February 2023. last accessed 07/03/2024. URL: https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/.

[14] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. Re-Factoring Based Program Repair Applied to Programming Assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 388–398, November 2019. ISSN: 2643-1572. URL: https://ieeexplore.ieee.org/abstract/document/8952522, doi:10.1109/ASE.2019.00044.

[15] IEEE. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, December 1990. Conference Name: IEEE Std 610.12-1990. URL: https://ieeexplore.ieee.org/document/159342, doi:10.1109/IEEESTD.1990.101064.

[16] Project Jupyter, Douglas Blank, David Bourgin, Alexander Brown, Matthias Bussonnier, Jonathan Frederic, Brian Granger, Thomas Griffiths, Jessica Hamrick, Kyle Kelley, M Pacer, Logan Page, Fernando Pérez, Benjamin Ragan-Kelley, Jordan Suchow, and Carol Willing. nbgrader: A Tool for Creating and Grading Assignments in the Jupyter Notebook. *Journal of Open Source Education*, 2(11):32, January 2019. URL: https://jose.theoj.org/papers/10.21105/jose.00032, doi:10.21105/jose.00032.

[17] Mohd. Ehmer Khan and Farmeena Khan. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. In *International Journal of Advanced Computer Science and Applications*, pages 22–25, June 2012.

[18] Natalie Kiesler and Daniel Schiffner. Large Language Models in Introductory Programming Education: ChatGPT's Performance and Implications for Assessments, August 2023. arXiv:2308.08572 [cs]. URL: http://arxiv.org/abs/2308.08572.

[19] Holger Krekel. pytest: helps you write better programs - pytest documentation, June 2024. last accessed 06/03/2024. URL: https://docs.pytest.org/en/8.2.x/.

[20] Angelo Kyrilov and David C Noelle. DO STUDENTS NEED DETAILED FEEDBACK ON PROGRAMMING EXERCISES AND CAN AUTOMATED ASSESSMENT SYSTEMS PROVIDE IT? 2016. URL: https://sites.ucmerced.edu/files/dnoelle/files/kyrilov-noelle-2016.pdf.

[21] Wladimir Iossifowitsch Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics-Doklady*, 10(8), February 1966.

[22] David Liu. PythonTA documentation, 2023. URL: https://www.cs.toronto.edu/~david/pyta/index.html.

[23] David Liu, Jonathan Calver, and Michelle Craig. A Static Analysis Tool in CS1: Student Usage and Perceptions of PythonTA. In *Proceedings of the 26th Australasian Computing Education Conference*, ACE '24, pages 172–181, New York, NY, USA, January 2024. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3636243.3636262, doi:10.1145/3636243.3636262.

[24] Hamza Manzoor, Amit Naik, Clifford A. Shaffer, Chris North, and Stephen H. Edwards. Auto-Grading Jupyter Notebooks. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 1139–1144, Portland OR USA, February 2020. ACM. URL: https://dl.acm.org/doi/10.1145/3328778.3366947, doi:10.1145/3328778.3366947.

[25] Leo A. Meyerovich and Ariel S. Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 1–18, Indianapolis Indiana USA, October 2013. ACM. URL: https://dl.acm.org/doi/10.1145/2509136.2509515, doi:10.1145/2509136.2509515.

[26] MongoDB Inc. MongoDB Database Scaling, June 2024. last accessed 06/09/2024. URL: https://www.mongodb.com/resources/basics/scaling.

[27] OpenAI. OpenAI Platform, June 2024. last accessed 06/12/2024. URL: https://platform.openai.com.

[28] Oracle. Java Development Kit Version 22 API Specification, June 2024. last accessed 06/30/2024. URL: https://docs.oracle.com/en/java/javase/22/docs/api/index.html.

[29] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, March 2016. URL: https://linkinghub.elsevier.com/retrieve/pii/S0306437915001611, doi:10.1016/j.is.2015.08.004.

[30] Joao Felipe Pimentel. APTED documentation, November 2017. last accesses 05/31/2024. URL: https://github.com/JoaoFelipe/apted.

[31] Project Jupyter. Extensions — JupyterLab 4.2.2 documentation, June 2024. last accesses 06/28/2024. URL: https://jupyterlab.readthedocs.io/en/stable/user/extensions.html.

[32] Project Jupyter. Project Jupyter, June 2024. last accessed 07/23/2024. URL: https://jupyter.org.

[33] Python Software Foundation. ast — Abstract Syntax Trees — Python 3.12.3 documentation, May 2024. last accessed 05/13/2024. URL: https://docs.python.org/3/library/ast.html.

[34] Python Software Foundation. Built-in Functions, June 2024. last accesses 06/30/2024. URL: https://docs.python.org/3/library/functions.html.

[35] Python Software Foundation. Installing Packages - Python Packaging User Guide, June 2024. last accessed 06/30/2024. URL: https://packaging.python.org/en/latest/tutorials/installing-packages/.

[36] Python Software Foundation. multiprocessing, June 2024. last accessed 06/02/2024. URL: https://docs.python.org/3/library/multiprocessing.html.

[37] Python Software Foundation. Python 3.12.4 Documentation, June 2024. last accesses 06/30/2024. URL: https://docs.python.org/3/.

[38] Python Software Foundation. The Python Language Reference, June 2024. last accessed 06/30/2024. URL: https://docs.python.org/3/reference/index.html.

[39] Python Software Foundation. The Python Standard Library, June 2024. last accesses 30/06/2024. URL: https://docs.python.org/3/library/index.html.

[40] Python Software Foundation. subprocess, June 2024. last accessed 06/04/2024. URL: https://docs.python.org/3/library/subprocess.html.

[41] Python Software Foundation. unittest, June 2024. last accessed 06/03/2024. URL: https://docs.python.org/3/library/unittest.html.

[42] Kelly Rivers and Kenneth R. Koedinger. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education*, 27(1):37–64, March 2017. URL: http://link.springer.com/10.1007/s40593-015-0070-z, doi:10.1007/s40593-015-0070-z.

[43] Valerie J. Shute. FOCUS ON FORMATIVE FEEDBACK. *ETS Research Report Series*, 2007(1), June 2007. URL: https://onlinelibrary.wiley.com/doi/10.1002/j.2333-8504.2007.tb02053.x, doi:10.1002/j.2333-8504.2007.tb02053.x.

[44] Draylson M. Souza, Katia R. Felizardo, and Ellen F. Barbosa. A Systematic Literature Review of Assessment Tools for Programming Assignments. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, pages 147–156, April 2016. ISSN: 2377-570X. URL: https://ieeexplore.ieee.org/document/7474479/?arnumber=7474479, doi:10.1109/CSEET.2016.48.

[45] The PEP Editors. PEP 0 – Index of Python Enhancement Proposals (PEPs) | peps.python.org, July 2024. last accesses 07/21/2024. URL: https://peps.python.org/pep-0000/.

[46] @tiangolo. FastAPI, 2024. last accesses 05/13/2024. URL: https://fastapi.tiangolo.com/.

[47] Guido van Rossum, Barry Warsaw, and Coghlan. PEP 8 – Style Guide for Python Code, July 2001. last accessed 05/13/2024. URL: https://peps.python.org/pep-0008/.

[48] Virginia Tech. What is Web-CAT? - Web-CAT, July 2024. last accessed 07/17/2024. URL: https://web-cat.org/projects/Web-CAT/WhatIsWebCat.html.

[49] Marvin Vogt. Analyses for Python Program Equivalences. May 2023.

[50] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT, February 2023. arXiv:2302.11382 [cs]. URL: http://arxiv.org/abs/2302.11382.

[51] Juliette Woodrow, Ali Malik, and Chris Piech. AI Teaches the Art of Elegant Coding: Timely, Fair, and Helpful Style Feedback in a Global Course. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, pages 1442–1448, Portland OR USA, March 2024. ACM. URL: https://dl.acm.org/doi/10.1145/3626252.3630773, doi:10.1145/3626252.3630773.

[52] Xingjiao Wu, Luwei Xiao, Yixuan Sun, Junhang Zhang, Tianlong Ma, and Liang He. A survey of human-in-the-loop for machine learning. *Future Generation Computer Systems*, 135:364–381, October 2022. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167739X22001790, doi:10.1016/j.future.2022.05.014.

[53] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. Repairing Bugs in Python Assignments Using Large Language Models, September 2022. arXiv:2209.14876 [cs]. URL: http://arxiv.org/abs/2209.14876.

[54] Stanislav Zmiev. AutoGrader Documentation, July 2024. last accesses 07/03/2024. URL: https://zmievsa.github.io/autograder/#/?id=installation.

Appendix B Evaluation Codes

```
def average_price_petrol(*Name, **Preis):
    summe=0
    for i in Preis:
        summe += Preis[i]
    avg= summe/len(Preis)
    return round(avg,4)
```

Listing B.1: Average Price 1

```
def average_price_petrol(**kwargs):
    price_list=[]
    price_total = 0
    for station, price in kwargs.items():
        price_list.append(price)
        #print (price_list)
    for item in price_list:
        price_total = price_total + item
    average_price = round(price_total / len(price_list), 3)
    print('The average petrol_price is:', average_price)
```

Listing B.2: Average Price 2

```
cars=dict()
cars['VW']={'Golf','Polo','UP!'}
cars['Seat']={'Leon','Ibiza'}
cars['Daihatsu']={'Sirion'}
cars['Audi']={'A4','Q3','R7','Q2'}
```

Listing B.3: Car Dictionary 1

```
cars = { "VW" : ("Golf", "Polo", "up!"),
        "Seat": ("Leon", "Ibiza"),
        "Daihatsu" : ("Sirion"),
        "Audi" : ("A4", "Q3", "R7", "Q2")}
```

Listing B.4: Car Dictionary 2

```
cars = dict()
cars["VW"] = set(["Golf, Polo, up!"])
cars["Seat"] = set(["Leon, Ibiza"])
cars["Daihatsu"] = set(["Sirion"])
cars["Audi"] = set(["A4, Q3, R7, Q2"])
```

Listing B.5: Car Dictionary 3

```
import random as rd

def create_random_list(size, min_value, max_value):
    list = []

    while len(list) < size:
        list.append(rd.randint(min_value, max_value))
    print(list)
```

Listing B.6: Random List 1

```
class Vehicle:

    def __init__(self, colour, construction_year, mileage):
        self.colour = colour
        self.construction_year = construction_year
        self.mileage = mileage

    def __str__(self):
        return f'The vehicle is {self.colour}, was manufactured in the year
                {self.construction_year} and has run {self.mileage}
                kilometres so far.'

    def drive (self, distance):
        self.mileage += distance
```

Listing B.7: Vehicle Class 1

```
class Vehicle:

    def __init__(self, mileage, colour, construction_year):
        self.colour = colour
        self.construction_year = construction_year
        self.mileage = mileage

    def drive(self, distance):
        self.mileage += distance

    def __str__(self):
        return f"The {self.__class__.__name__} is {self.colour}, was
                manufactured in the year {self.construction_year} and has run
                {self.mileage} kilometres so far."
```

Listing B.8: Vehicle Class 2

Appendix C Application Programming Interface

Table C.2 shows a brief overview of the available Application Programming Interface (API) routes offered by the web interface, table C.1 of those offered by the check and repair container. A more detailed overview is available using the Swagger documentation in the digital appendix D.

Type	URL	Description
POST	/check	Run a given list of checks on the provided code
POST	/cluster	Cluster the provided code using the provided reference solutions
POST	/repair	Run code repair on the provided code using the provided reference solutions
POST	/typecheck	Check if a given literal is of a given type
Type	**URL**	**Description**

Table C.1: API endpoints of the check and repair container

Type	URL	Description
DELETE	/api/admin/ check-templates/{id}	Delete a check template
PUT	/api/admin/ check-templates/{id}	Edit a check template
POST	/api/admin/ check-templates	Add a check template
PUT	/api/admin/checks/ {id}/down	Move a check down
DELETE	/api/admin/checks/ {id}	Delete a check
PUT	/api/admin/checks/ {id}	Edit a check
PUT	/api/admin/checks/ {id}/up	Move a check up
Type	**URL**	**Description**

Table C.2: API endpoints of the web interface (continued in table C.3)

Type	URL	Description
PUT	/api/admin/checks/{id}/verify	Mark a check as verified, arg types are checked
POST	/api/admin/checks/convention-hints	Generate convention hints for a task
POST	/api/admin/checks/generate	Generate checks for a task using OpenAI
POST	/api/admin/tasks/checks	Create a check
DELETE	/api/admin/exercises/{id}	Delete exercise
PUT	/api/admin/exercises/{id}	Update exercise
POST	/api/admin/exercises	Create a new exercise
POST	/api/admin/export	Export exercises, tasks, and check templates
POST	/api/admin/import	Import data stored in a JSON file, including tasks, exercises, and check templates
DELETE	/api/admin/solutions/{id}	Delete a solution
PUT	/api/admin/solutions/{id}	Update a solution
PUT	/api/admin/solutions/{id}/promote	Promote a new solution to a reference solution
PUT	/api/admin/solutions/{id}/unvalidate	Mark a solution as "not validated" by removing the "new" flag
POST	/api/admin/solutions/generate	Generate a solution for a task using the OpenAI api
POST	/api/admin/solutions	Create a new solution
POST	/api/admin/solutions/validate	Validate a code snippet against a task's unit tests
PUT	/api/admin/tasks/{id}/cluster	Update the cluster settings of a task
DELETE	/api/admin/tasks/{id}	Delete a task
PUT	/api/admin/tasks/{id}	Update a task
PUT	/api/admin/tasks/{id}/repair	Update the repair settings of a task
POST	/api/admin/tasks	Create a new task
Type	**URL**	**Description**

Table C.3: API endpoints of the web interface (continuing table C.2)

Appendix D Digital Appendix

The digital appendix is submitted on a Micro SD card. To get access to this material, please contact the author or LuFG i9.

Eidesstattliche Versicherung

Steffes, Leonard	394539
Name, Vorname	Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Masterarbeit mit dem Titel

Contextual Automatic Code Repair for Python Programming Novices

ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen , August 5, 2024	
Ort, Datum	Unterschrift

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Aachen , August 5, 2024	
Ort, Datum	Unterschrift