Revista Brasileira de Informática na Educação – RBIE
Brazilian Journal of Computers in Education
(ISSN online: 2317-6121; print: 1414-5685)
https://sol.sbc.org.br/journals/index.php/rbie

Submission: 17/Jul/2023;
Camera ready: 07/Dec/2023;
1st round notif.: 26/Nov/2023;
Edition review: 27/Dec/2023;
Available online: 27/Dec/2023;
Published: 27/Dec/2023;

When Test Cases Are Not Enough: Identification, Assessment, and Rationale of Misconceptions in Correct Code (MC³)

Eryck Pedro da Silva
Universidade Estadual de Campinas (UNICAMP)
ORCID: 0000-0001-5141-6936
eryck.silva@ic.unicamp.br

Ricardo Caceffo
Universidade Estadual de Campinas (UNICAMP)
Virtual University of São Paulo (Univesp)
ORCID: 0000-0001-9152-0658
caceffo@ic.unicamp.br
ricardo.caceffo@univesp.br

Rodolfo Azevedo
Universidade Estadual de Campinas (UNICAMP)
ORCID: 0000-0002-8803-0401
rodolfo@ic.unicamp.br

Abstract
Automated grading systems (autograders) assist the process of teaching in introductory programming courses (CS1). However, the sole focus on correctness can obfuscate the assessment of other characteristics present in code. In this work, we investigated if code, deemed correct by an autograder, were developed with characteristics that indicated potential misunderstandings of the concepts taught in CS1. These characteristics were named Misconceptions in Correct Code (MC³). By analyzing 2,441 codes developed by CS1 students, we curated an initial list of 45 MC³. This list was assessed by CS1 instructors, resulting in the identification of MC³ that should be addressed in classes. We selected the 15 most severe MC³ for further investigation, including a semi-structured observation in a CS1 course and an automated detection software using static code analysis. The results suggested that students develop these MC³ either due to an incomplete comprehension of the concepts taught in CS1 course or a lack of attention while elaborating their code, with correctness being their primary goal. We believe our results can contribute to: (1) the research field of misconceptions in CS1; (2) promoting alternative approaches to complement the use of autograders in CS1 classes; and (3) providing insights that can serve as the foundation for teaching interventions involving MC³ in CS1.

Keywords: *Introductory Programming; Misconceptions; Automated Assessment; Autograders; CS1.*

1 Introduction

The need for interdisciplinary domains to solve modern problems constantly requires professionals with knowledge in computer literacy. As a result, an increasing number of undergraduate programs are incorporating Computer Science (CS) related classes into their curricula. While these classes are predominantly found in programs within the Science, Technology, Engineering, and Mathematics (STEM) field (Brodley et al., 2022; Dodds et al., 2010), they are also being introduced in other areas, including Law (Sloan et al., 2017).

When considering the context of CS teaching, one of the common courses focuses on introductory programming concepts, commonly referred to as CS1 (Austing et al., 1979). CS1 courses are often associated with challenges, including high failure and dropout rates (Bosse & Gerosa, 2015; Kinnunen & Malmi, 2006; Walker, 2017). Among the various factors contributing to these rates, the number of students per class stands out. The high student-to-teacher ratio hinders individual attention, leading to student demotivation.

Automated grading systems, commonly known as autograders, have been employed in CS1 courses to address the challenges associated with large class sizes. As early as 1960, Hollingsworth (1960) highlighted the potential benefits of autograders, including time and cost savings, as well as the ability to accommodate larger classes. Today, autograders are extensively used in courses with these characteristics, particularly in Massive Open Online Courses, facilitating individual attention between instructors and students (Marwan et al., 2019).

In the educational context, autograders play a crucial role in grading assignments, relieving instructors of some of their workload and conserving resources (Ellis et al., 2019; Galvão et al., 2016; Hollingsworth, 1960). However, the use of autograders can also influence the development of undesirable habits in students. Baniassad et al. (2021) noted that students may become overly reliant on the feedback provided by autograders, leading them to rely on trial-and-error approaches in their coding. Frustration among CS1 students can also arise when automated tools experience malfunctions (Inside Higher Ed, 2018) or when students mistakenly assume that autograders cannot make mistakes, resulting in overconfidence (Hsu et al., 2021).

One common application of autograders in CS1 courses is to verify if the output of a code matches the expected predetermined output (Prather et al., 2018). This approach fosters research aimed at understanding and enhancing the teaching and learning of concepts based on code correctness (L. G. Araujo et al., 2021; Becker et al., 2018; Liu & Petersen, 2019; Pereira, Oliveira, et al., 2020). However, in CS1, even code that produces the desired outcome can exhibit undesirable characteristics that experienced programmers would typically avoid (De Ruvo et al., 2018; Soloway & Ehrlich, 1984). Examples of such characteristics include higher code complexity resulting from the redundant use of syntactic constructs, such as conditional statements or nested loops (Ihantola & Petersen, 2019; Silva et al., 2021; Ureel II & Wallace, 2019). The sole focus on code correctness may lead students to disregard other important code attributes, such as readability and maintainability, which are crucial for future programmers (De Ruvo et al., 2018; Keuning et al., 2019). In a CS1 setting where both instructors and students solely assess code functionality without considering other indicators, potential misconceptions or incomplete understanding of CS1 topics may go unnoticed and unaddressed.

An example of incomplete understanding of a CS1 topic is illustrated in Code 1. In this code, a student employed conditional statements to determine whether the value of the variable **var** was zero/positive or negative. However, likely due to a belief that the **else** clause is mandatory, the student checked both desired cases for **var** in lines 2 and 4, while also including an unnecessary **else** statement in line 6. Despite knowing that both necessary conditions were already checked in the **if-elif** statements, the student mistakenly believed that the **else** clause was obligatory and included redundant instructions within its body to not alter the correct output. Code 2 provides an alternative implementation that resolves the misconception demonstrated in Code 1.

```
1  var = int(input())
2  if var >= 0:
3      print("var␣is␣positive␣or␣0")
4  elif var < 0:
5      print("var␣is␣negative")
6  else:
7      (...)
```

Code 1: Example of unnecessary **else**.

```
var = int(input())
if var >= 0:
    print("var␣is␣positive␣or␣0")
else:
    print("var␣is␣negative")
```

Code 2: Code 1 without the unnecessary **else**.

This work was motivated by the need to assess code that had already been deemed correct by an autograder. By *correct*, we refer to code that successfully passed all the tests designed to evaluate the code output for specific inputs. Since we were within a CS1 context, these tests primarily focused assessing the correctness of the code by testing both standard and boundary values as inputs. In this work, we were specifically interested in identifying whether correct code could exhibit behaviors that potentially indicated incomplete understanding of CS1 topics. To conduct our analysis, we classified these behaviors as misconceptions (Qian & Lehman, 2017). In CS1, research on misconceptions typically focuses on identifying and classifying errors made by students, including syntactic, semantic, or logical errors (A. Araujo et al., 2021; Caceffo et al., 2016, 2019; Gama et al., 2018). However, these studies are not necessarily limited to correct code. Therefore, we chose to narrow our investigation and specifically examine code that produced the expected results. As a result, we established a subgroup called Misconceptions in Correct Code (MC^3). In other words, whereas MC^3 are misconceptions within the CS1 research field, not all misconceptions studied in this field can be classified as MC^3.

With the hypothesis that MC^3 exist in code deemed correct by an autograder, this work aimed to address the following research questions:

RQ1: *Which MC^3 are the most severe, requiring high-priority explanations in CS1 courses?*

RQ2: *How can MC^3 be potentially addressed in CS1 classes, considering multiple contexts of teaching and learning?*

RQ3: *What is the frequency distribution of MC^3 in a typical CS1 course?*

RQ4: *What are the reasons behind CS1 students incorporating MC^3 into their code?*

In total, an exploratory analysis of students' code submitted for assignments in a CS1 course revealed the presence of 45 MC^3, indicating misconceptions and incomplete understandings of key CS1 topics. To prioritize the most critical misconceptions for classroom intervention, a survey was conducted among CS1 instructors. From this survey, the top 15 MC^3 were identified and given priority to investigation. These misconceptions predominantly revolved around Boolean expressions and iteration, with additional focus on code organization, the use of variables and functions, and the characteristics of autograders, including test cases. The analyses involving CS1 instructors and students shed light on the reasons behind the incorporation of MC³ in code, which included incomplete comprehension of CS1 topics and a lack of attention to coding practices, stemming from a narrow focus on correctness alone. CS1 instructors and students also emphasized that addressing MC³ in CS1 classes can be facilitated through automated detection and feedback mechanisms, integration into lecture classes, and the adoption of Active Learning techniques. Additionally, we developed a prototype of an automated detection tool for the most severe MC³, and while the occurrence of these misconceptions was not found to be high in absolute numbers, their presence was observed throughout the entirety of a CS1 course.

We believe that our findings can contribute for the broader community, particularly in the context of CS1 education supported by autograders from which MC³ are possibly being overlooked. While previous literature may have identified similar behaviors to MC³, our survey with CS1 instructors and conversations with students provided additional insights into the underlying reasons for these misconceptions throughout the course. Based on the evidence we have gathered, we advocate for the development of formative feedback that can be directed towards instructors, teaching assistants, and students, with the aim of enhancing the teaching and learning experience in CS1 courses.

The remainder of this paper is organized as follows. Section 2 presents the background and related work. The methodology used are described in Section 3, followed by the obtained results in Section 4. We discuss the results in Section 5. Section 6 details the limitations and threats to validity of this research. Lastly, we present the conclusions in Section 7.

2 Background and Related Work

In this section, we dedicate our focus to providing a theoretical background that formed the basis of our hypothesis to the development of this work, as well as discussing related research in a similar domain. The background section delves into the intricate details of student errors commonly encountered in CS1, as well as highlighting the role of autograders in this context. Subsequently, we describe related works with the aim of examining and synthesizing existing literature that has addressed similar challenges, while also identifying their strategies and methodologies. Additionally, we clarify the specific contribution and position of our work within this broader research landscape.

2.1 Background

There are various terms used to describe faulty comprehensions of concepts taught in CS1 classes. Qian and Lehman (2017) conducted a systematic literature review on this topic and identified terms such as *errors*, *bad comprehensions*, *challenges*, and *misconceptions* commonly used in the literature. They classified these faulty comprehensions into different levels of knowledge: syntactic, conceptual, and strategic. At the syntactic level, comprehension issues arise from a lack of understanding of basic rules of a programming language, such as mandatory Python indentation or the use of semicolons in Java. The conceptual level encompasses issues related to the understanding of programming constructs, such as variable declaration or loops. Finally, issues at the strategic level occur when students struggle to apply the knowledge acquired at the syntactic and conceptual levels while solving problems. Qian and Lehman classified misconceptions as issues present in the conceptual level.

The identification of misconceptions is an important aspect of developing concept inventories (CI) (Ali et al., 2023; Almstrum et al., 2006). A CI is an assessment tool specifically designed to identify and address misconceptions within a specific domain of knowledge (Almstrum et al., 2006; Caceffo et al., 2016), often in the form of a multiple-choice questionnaire. Almstrum et al. (2006) proposed a development process for constructing and validating a CI, which involves steps such as using open-ended questions to discover misconceptions, interviewing students, piloting a set of multiple-choice questions, and employing statistical analysis to validate the CI. This development process has been applied to the creation of CI for CS1 in programming languages such as C (Caceffo et al., 2016), while there are ongoing research for Python (Gama et al., 2018) and Java (Caceffo et al., 2019). Additionally, Tew and Guzdial (2011) developed the Foundational CS1 (FCS1) assessment tool, designed to be used independently of a specific programming language.

Regarding automated assessment tools, Ureel II and Wallace (2019) identified two distinct groups that these tools may fall into: autograders and critiquers. Autograders primarily focus on unit testing and may not be suitable for providing feedback on all types of bad coding behaviors, as some of them might happen on correct code. On the other hand, critiquers are similar to autograders, but aim to provide feedback based on the instructors' pedagogical knowledge. This formative feedback is crucial as it can lead to significant improvements in students' understanding (Cain & Babar, 2016). However, there is a risk of students becoming overly reliant on automated feedback (Baniassad et al., 2021). Baniassad et al. (2021) found that students were using the feedback merely to correct mistakes in their code without engaging in thorough thinking. To address this issue, the authors implemented a penalty system in a CS1 course, whereby students received lower grades for successive submissions to an autograder. As a result, the authors observed a decrease in the number of submissions while only slightly affecting the median grade. Although the students expressed concerns with each submission, they also reported that they checked their code more carefully and analyzed their mistakes before submitting again.

Instructors and teaching assistants can also benefit from the feedback provided by autograders or similar tools. Pereira et al. (2020) conducted a study where they collected and analyzed various features from students' submissions to a CS1 course in Python, enabling them to predict whether students would pass or fail the course. These features included, but were not limited to, the number of submissions, time spent on each submission, number of problems solved correctly, and average lines of code per submission. The authors argued that assessing the first two weeks of assignments is crucial for creating an early prediction and providing support to students who may be at risk of failing the course. Similarly, using machine learning techniques, Lima et al. (2021) classified coding questions present in an autograder to ensure a balanced distribution of assignments among students. To achieve this, the authors analyzed past students' submissions to these questions and collected code attributes such as complexity and the number of syntax constructs, along with the success rate in completing the assignments. Both studies were conducted using CodeBench[1], an autograder developed by the Federal University of Amazonas.

[1]https://codebench.icomp.ufam.edu.br/

2.2 Related Work

By analyzing students' answers to exams and interviewing CS1 instructors, Caceffo et al. (2016) identified 15 misconceptions in the C programming language. These misconceptions were classified into seven categories: function parameter use and scope; variables, identifiers, and scope; recursion; iteration; structures; pointers; and Boolean expressions. The findings from this analysis served as the basis for the development of a concept inventory.

Gama et al. (2018) conducted an analysis to examine the applicability of the misconceptions identified by Caceffo et al. (2016) to the Python language. Based on the frequency of these misconceptions in open-ended exam questions, they made decisions regarding whether to retain or discard each misconception. Two categories, structures and pointers, were deemed irrelevant in Python and were discarded, while a new category emerged: use and implementation of classes and objects. In total, Gama et al. hypothesized 28 misconceptions, requiring further validation.

Araújo et al. (2021) expanded upon the results obtained by Gama et al. (2018) through an empirical study. Given the similarity in the teaching contexts of the analyzed CS1 courses in both works, the authors stated that this empirical study could be conducted. Similar to the previous studies, students' answers to open-ended exam questions were used. Araújo et al. identified 27 misconceptions, with 19 of them present in the listing provided by Gama et al. The remaining eight misconceptions were grouped into a new category called Additional, which consisted of simple logic and syntactic errors.

Regarding the use of autograders in CS1, Araujo et al. (2021) developed the Python Enhanced Error Feedback (PEEF). PEEF is an online integrated development environment (IDE) that provides enhanced compiler error messages, an integrated chat feature, and performs dynamic code analysis through unit testing. The authors discussed the potential uses of this tool for both students and instructors in CS1 courses. Another tool, PyTA, was created by Liu and Petersen (2019). PyTA promotes static code analysis (Wichmann et al., 1995) to provide comprehensive feedback by presenting warnings and error messages in a simplified manner to students. Liu and Petersen conducted a study in which students had the option to consult or not consult this enhanced feedback. Among the students who chose to use it, they observed a reduction in the number of errors per assignment, the total number of submissions until an assignment had an error corrected, and the total number of submissions until the assignment passed all the test cases.

Among research that focused on analyzing correct code in CS1, De Ruvo et al. (2018) introduced the concept of semantic styles. Semantic styles are indicators that potentially reveal a poor understanding of programming concepts. The authors analyzed students' submissions to programming assignments and identified 16 semantic styles. Among these, 12 were related to conditional commands, such as unnecessary else statements or duplicated code within an if/else structure. The remaining four semantic styles were associated with the use of variables.

Motivated by the goal of providing formative feedback that closely resembles that of an instructor, Ureel II and Wallace (2019) developed WebTA. They classified their tool as an automated critiquer capable of detecting anomalous coding behaviors, regardless of whether the code is correct. WebTA can identify pre-existing misconceptions from the literature (approximately 200), and it also allows instructors to create new coding behavior rules to detect specific anomalous code expected for an assignment.

The A-Learn Evid, developed by Porfirio et al. (2021), is an automatic method for identifying students' programming skills. The authors aimed to automate the assessment of these skills, going beyond functionality, in order to allow instructors to provide timely and formative feedback to students. The method employs both static and dynamic analysis of students' source code and can identify 37 programming skills. Examples of these skills include variables, Boolean expressions, infinite loops, control structures, and functions.

Refactoring Programming Tutor (RPT)[2], developed by Keuning et al. (2021), is an Intelligent Tutoring System (VanLehn, 2006) that provides step-by-step hints for improving the quality of correct code. The system incorporates rules idealized by experienced instructors, refactoring rules found in established software, and previous literature, including the semantic styles identified by De Ruvo et al. (2018). Keuning et al. outlined 19 refactoring rules implemented in RPT, covering areas such as expressions, branching, loops, and declarations.

In their study, Oliveira et al. (2023) analyzed program snapshots of students who worked on programming exercises in RPT to identify errors made by students during code refactoring. The authors examined 482 sequences of these program snapshots, which were created by 133 students. Based on their analysis, Oliveira et al. categorized these errors as refactoring misconceptions. They identified a total of 25 refactoring misconceptions, which were catalogued into five groups: arithmetic expressions, Boolean expressions, conditionals, flow, and loops.

Table 1 presents a comparison of our work with the related research discussed in this section. One key characteristic of our work is its focus on analyzing misconceptions exclusively in code that is deemed correct by an autograder, which sets it apart from some of the related research that did not apply this condition (A. Araujo et al., 2021; L. G. Araujo et al., 2021; Caceffo et al., 2016; Gama et al., 2018; Liu & Petersen, 2019). Additionally, our study specifically targets misconceptions found in students' code for CS1 courses taught in Python, while other research had primarily focused on different programming languages (De Ruvo et al., 2018; Keuning et al., 2021; Oliveira et al., 2023; Ureel II & Wallace, 2019).

Table 1: Comparison of this research with the presented related work.

Research	Errors/Misconceptions analyzed	Analysis of correct code	Language
(Caceffo et al., 2016)	15	-	C
(Gama et al., 2018)	28	-	Python
(A. Araujo et al., 2021)	21	-	Python
(L. G. Araujo et al., 2021)	Not mentioned*	-	Python
(Liu & Petersen, 2019)	Not mentioned*	-	Python
(De Ruvo et al., 2018)	16	✓	Java
(Ureel II & Wallace, 2019)	200	✓	Java
(Porfirio et al., 2021)	37	✓	C
(Keuning et al., 2021)	19	✓	Java
(Oliveira et al., 2023)	25	✓	Java
This research	45	✓	Python

*Total not informed. Authors used enhanced compiler error messages for Python errors.

[2]http://hkeuning.nl/rpt/

3 Methods

In this section, we describe the methods employed in the research on MC³, encompassing data collection and analysis. The section begins by providing background information on the analyzed CS1 course and outlining the identification process of the MC³. Subsequently, we present the details of the severity ranking (RQ1) and explain how the MC³ can be addressed in CS1 classes (RQ2). We then explain how the frequency of the most severe MC³ was calculated (RQ3), followed by how we delved into the reasons why students incorporate MC³ into their code (RQ4). Figure 1 summarizes the methods used in this work.

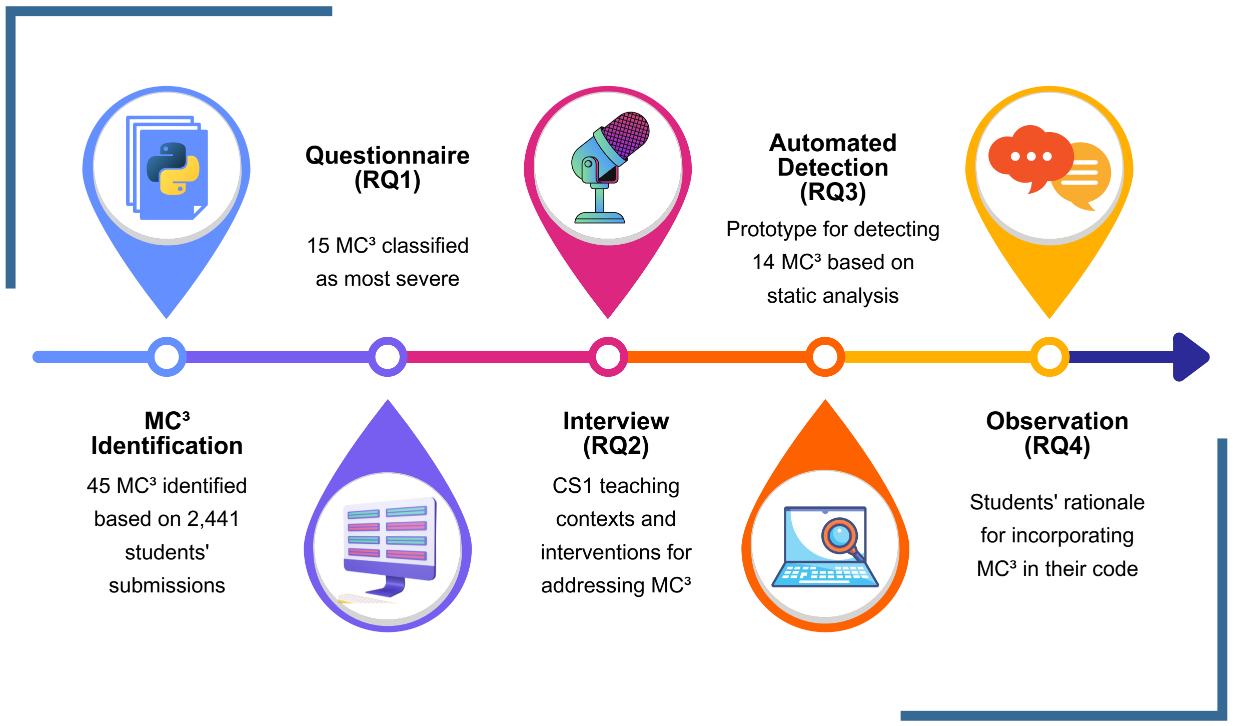
Questionnaire (RQ1)
15 MC³ classified as most severe
Automated Detection (RQ3)
Prototype for detecting 14 MC³ based on static analysis
MC³ Identification
45 MC³ identified based on 2,441 students' submissions
Interview (RQ2)
CS1 teaching contexts and interventions for addressing MC³
Observation (RQ4)
Students' rationale for incorporating MC³ in their code

Figure 1: Description of the methods used in this research.

3.1 MC³ Identification

The primary objective of this phase was to determine whether code deemed correct by an autograder exhibited characteristics that could indicate an incomplete understanding of CS1 learning objectives. To achieve this, we analyzed the course of Algorithms and Computer Programming (MC102) at UNICAMP, which has a high number of enrolled students per semester (approximately 600). MC102 is organized in a coordinated environment that follows the same syllabus and practical assignments to a group of bachelors' programs, mostly engineering. The course teaches the imperative paradigm using the Python programming language since 2018.

The basic syllabus for MC102 is the following: basic concepts of computer organization; data I/O; arithmetic, logical, and relational expressions; conditional commands; repetition commands; lists, tuples, dictionaries, strings, and matrices; functions and scope of variables; sorting algorithms; searching algorithms; recursion; and recursive sorting algorithms.

During the semester, MC102 students are assigned practical tasks. Although these tasks cover most of the syllabus topics, some topics are combined within a single one (e.g., lists and tuples, strings and dictionaries), while others lack dedicated tasks (e.g., functions and recursive sorting algorithms). All assignments are submitted via SuSy[3], an autograder developed within the institution itself. SuSy performs a dynamic analysis of each submission to verify whether the output matches the expected results for each task. This assessment is conducted using test cases, which consist of predefined input and expected output data. Test cases can be open, which are visible to students, whereas closed test cases are not visible. The grade for each assignment is determined by the number of test cases the students' submissions pass. SuSy may also limit the maximum number of submissions (typically set to 20 to prevent trial-and-error usage) and imposes a maximum execution time for the code. The system is also capable of detecting plagiarism among submissions. To identify the presence of MC³ in code deemed correct by an autograder, we collected and analyzed the students' submissions to MC102 assignments.

3.1.1 Data Collection

All students' submissions were collected using SuSy. We analyzed submissions from a total of 19 different bachelors' programs in the first term of 2020. The process of data collection and analysis took place after the term ended.

Since our objective was to analyze characteristics present in correct code, a filtering process had to be conducted before the analysis began. For each submission, SuSy generates a log file that contains various information, including the total number of test cases passed by the submission. The system retains only the last submission made by each student. By utilizing this log file, we were able to identify the code that passed all test cases for all assignments.

In the aforementioned academic term, the course had a total of 14 assignments. As this research was in its initial exploratory stages, it was decided to only collect tasks assigned in the first half of the course, before the first partial exam. This decision considered the identification of undesirable behaviors and incomplete comprehensions developed during the learning of basic CS1 topics. Our goal in doing this was because if these characteristics are not addressed early on, they may manifest in more complex topics taught later. Additionally, some assignments within this interval covered the same topic and were excluded from the analysis. In total, six assignments were chosen for analysis, as described in Section 4.

3.1.2 Data Analysis

All submissions were manually analyzed, following the sequential order of the assignments. We created spreadsheets to organize the occurrences identified in students' code. Initially, these occurrences were simple annotations that described coding behaviors present in the submissions. As the analysis progressed, we identified similar behaviors and assigned them provisional names, grouping and updating related occurrences as necessary. Since we planned to obtain external validation (RQ1 and RQ2) of these coding behaviors before conducting further investigations, we did not perform any assessment of the MC³ frequency.

[3]http://ic.unicamp.br/~susy/

After completing the analysis of all submissions, a categorization process was initiated. This process consolidated the MC³ by grouping similar occurrences that had been annotated. The categories were named based on the work of Gama et al. (2018). We analyzed a total of 2,441 submissions, resulting in an initial list of 45 MC³ divided into 8 categories, detailed in Section 4.

While it can be argued that the identification of MC³ was thorough because we analyzed submissions from 19 different bachelors' programs from the same CS1 course, there are potential issues to be noted. The discovered MC³ may be influenced by institutional locality since all programs were from the same institution. Furthermore, the interpretation of MC³ may be influenced by the researchers' bias. To address these issues, we conducted a survey involving CS1 instructors to assess the MC³. The survey consisted of an online questionnaire to classify the severity of each MC³, and a semi-structured interview (Lazar et al., 2017) to identify different teaching and learning contexts of CS1 and explore how MC³ could be addressed in classes. Due to the COVID-19 pandemic and the desire to reach a broader audience, the survey was conducted entirely online. As the survey involved human participants, it received prior evaluation and approval from a Research Ethics Committee[4].

3.2 RQ1: MC³ Severity Classification

Our objective was to determine how CS1 instructors would classify all 45 MC³ in terms of the severity of these coding behaviors. By severity, we refer to the high priority need for explanation in CS1 classes, as these MC³ indicate potential misconceptions or incomplete understandings of the learning objectives. Another anticipated outcome of this phase was to establish a ranking for the initial list of MC³. We believed that with a ranked list, we would be able to identify and further investigate the most severe MC³.

3.2.1 Data Collection

We collected the data using an online questionnaire. The invitation period spanned from January to February 2022, and we accepted responses until the end of March of the same year. We distributed the invitations through discussion lists and directly contacted authors who had recently published papers focused on CS1. The estimated average completion time for the questionnaire was between 40 to 55 minutes. The completion process involved the following steps:

1. Detailing the Informed Consent Form that provided a comprehensive explanation of the research and requested the respondent's consent.

2. Basic contextualization questions about the respondent, including name, institution of employment, years of experience teaching CS1, experience teaching Python, and familiarity with other programming languages.

3. Questions for classifying the severity of the MC³. Each item in the questionnaire presented the MC³ name, a brief description, a generic code sample illustrating the MC³, and a description of the sample. The severity classification consisted of two parts: a Likert item

[4]Approval can be consulted in Plataforma Brasil with CAAE number: 51444121.5.0000.5404.

inquiring whether the respondent considered the MC³ to be severe, and an optional text field for additional comments on the MC³.

4. Invitation for the respondent to participate in the semi-structured interview.

3.2.2 Data Analysis

The MC³ severity ranking was conducted based on the frequencies obtained from the Likert items. Initially, we grouped the frequencies of similar response categories: strongly disagree (SD) and disagree (D), neutral (N) and blank (B), and strongly agree (SA) and agree (A). Next, we calculated the difference between the frequencies of those who considered the MC³ to be severe and those who did not. This difference, referred to as DIF, can be interpreted as $(SA + A) - (N + B + SD + D)$.

The commentaries provided by respondents were analyzed using context analysis (Lazar et al., 2017). Four main topics related to the MC³ were identified: severity, frequency, reasons for their occurrence, and strategies to mitigate their occurrence. Although the results obtained with this analysis were not directly used in the ranking of MC³, we believe that the obtained data can contribute to a deeper understanding of these behaviors and serve as a foundation for further investigation into MC³.

3.3 RQ2: Addressing MC³ in CS1

We conducted semi-structured interviews (Lazar et al., 2017) with CS1 instructors to answer this question. The interviews aimed to gather more information about the diverse contexts of teaching and learning of CS1, explore whether the MC³ or similar behaviors occur in other CS1 courses, and understand how instructors handle these behaviors. Additionally, we sought to gather instructors' opinions on potential interventions to address the MC³ in CS1 classes.

3.3.1 Data Collection

Since this research on MC³ was primarily exploratory in nature, we chose a semi-structured format for the interviews to allow for flexibility. To ensure geographic diversity among the CS1 instructors who had volunteered for the interviews, we chose to invite at least one instructor from each institution in every participating country. The interviews took place between March and July of 2022 and were expected to last approximately 40 minutes each. We utilized Google Meet as the platform to conduct the conversations. The interviewing process consisted of the following steps:

1. A brief introduction by the researcher, including an explanation of the research purpose and objectives, followed by a request for the interviewee's consent to record the interview.

2. A set of questions focused on the structure of the CS1 classes taught by the instructor.

3. A set of questions concerning the MC³ and how the instructor handles them.

4. A set of questions related to teaching and learning interventions aimed at mitigating the occurrence of MC³ and the instructor's perspectives on their implementation in the classroom.

3.3.2 Data Analysis

After the interviews were concluded, each answer was compiled and analyzed individually using content analysis (Lazar et al., 2017). The information obtained from the interviews was organized into different categories, which included the context in which the instructors teach, such as the class outline, assigned tasks, and the use of autograders. Additionally, the instructors' opinions on the MC³ were examined, including whether they observed these behaviors in their classes, how they dealt with them, and examples of other similar behaviors they encountered. Furthermore, the instructors provided insights on potential artifacts and interventions to address the MC³ in CS1 classes, such as the use of autograders capable of detecting MC³ and the implementation of Active Learning (Bonwell & Eison, 1991) techniques specifically targeting these coding behaviors.

3.4 RQ3: Frequency Distribution of MC³

Building upon the insights obtained from RQ1 and RQ2, our next endeavor was to explore the automatic detection of the most severe MC³. Automating this procedure would not only facilitate its integration with an autograder but also provide the opportunity to assess the frequency of MC³ in our dataset, as well as in other datasets. Additionally, analyzing the distribution of MC³ occurrences would shed light on when these behaviors emerge in CS1, in terms of the topics covered in assignments, and whether they persist until the end of the course.

3.4.1 Data Collection

To accomplish the automatic detection, we leveraged static analysis techniques (Wichmann et al., 1995). Specifically, we employed Python module AST[5] for this purpose. This module provides tools to inspect and modify the Abstract Syntax Tree (Kluvyer, n.d.) of Python code, which is generated after parsing the syntax but before compiling the bytecode.

Out of the 45 identified MC³, a subset of 15 was deemed the most severe (refer to Section 4 for detailed information). We chose to prioritize the implementation of the automatic detection process for this subset. The implementation was carried out exclusively in Python 3, utilizing its AST module. We collected and analyzed students' submissions for the MC102 course that were elaborated in the first academic term of 2020 and the second term of 2022.

3.4.2 Data Analysis

The assignments in each analyzed academic terms were different from each other. To account for this, we opted to group them based on the CS1 concept intended to be explored in these assignments. For each submission, we parsed the code and checked for the presence of the selected MC³, counting whether the MC³ was present or not in the code. Subsequent occurrences of the same MC³ in the same code were not counted twice.

During the implementation process, we encountered certain challenges in the automated detection of certain MC³. Specifically, decisions regarding code that was deemed redundant, non-significant, or unnecessary still required instructor intervention. In such cases, we employed

[5]https://docs.python.org/3/library/ast.html

threshold values to determine if the code exhibited the associated MC³ or not. Further details on the grouping of CS1 concepts and the thresholds used can be found in Section 4.

3.5 RQ4: Why Students Code with MC³

During the identification of the MC³, we were unable to directly inquire with the students about why they incorporated these behaviors into their code, as the first term of 2020 had already concluded. While we did gather some insights on the reasons for MC³ occurrences through RQ1 and RQ2, those responses were provided by CS1 instructors, not students. Given this limitation, we decided to investigate one context of teaching and learning of CS1 to understand what could contribute to the development of MC³ by students. To achieve this, we conducted an assessment during an academic semester of the MC102 course in the second term of 2022 and in the first term of 2023, engaging with both students and the instructor. As with our previous methodology, this approach involving human participants underwent evaluation and approval by a Research Ethics Committee[6].

3.5.1 Data Collection

A semi-structured observation (Cohen et al., 2005) methodology was employed to gather the data. Cohen et al. (2005) suggest that this approach is ideal for capturing real-time data in live situations, accessing personal knowledge, and analyzing details that might have been overlooked. Given the phenomenological nature of the participants' experiences, we adopted a qualitative approach to explore the connections, causes, and correlations related to the MC³ over time.

Prior to analyzing the students' submitted code for the assignments, we obtained their consent to use their code for research purposes. The presence of MC³ in the code was examined after the submission deadline. In this research, all code submissions were checked for the presence of any MC³, regardless of the severity of the behaviors. Following each assignment analysis, one researcher conducted brief conversations of approximately 10 to 15 minutes, during which he asked the students about their reasons for incorporating MC³ into their code. Simultaneously, he explained the identified MC³ to them. These conversations took place outside of class hours, at a time and place mutually agreed upon by the student and the researcher. The observational data was recorded by the instructor using field notes.

In a similar manner, the instructor was asked for his consent to be observed during his classes. The same researcher attended all lectures during the second term of 2022, each one being a 2 hour slide based class with some code examples executed. The primary objective of this observation was to analyze whether the educational material used in the classes might potentially contain MC³, which could influence students to develop these behaviors in their code. Analogous to the conversations, the researcher also recorded observational data from the CS1 classes using field notes. At the end of the term, the researcher conducted an interview with the instructor to present our findings.

[6]Approval can be consulted in Plataforma Brasil with CAAE number: 60258622.8.0000.5404.

3.5.2 *Data Analysis*

The field notes that were developed during the observation of the instructor and the conversations with students were analyzed using content analysis (Lazar et al., 2017). By observing the lectures, the researcher aimed to identify whether the educational materials utilized, such as class slides or code developed in class, contained any instances of MC³. At the conclusion of the term, all identified MC³ in these materials were cataloged. Similarly, we analyzed the various reasons provided by the students for incorporating MC³ into their code, grouping together similar explanations. While we investigated all instances of MC³ that occurred in both the classroom and the conversations, for the purpose of this paper, we will focus solely on the most severe behaviors as identified in RQ1.

4 Results

This section presents the results obtained, following the same order as described in Section 3. Firstly, we provide a detailed explanation of the initial list of identified MC³ and their respective categories, along with the assignments that were analyzed during this phase. Next, we present the results obtained from the online questionnaire, including the severity ranking of MC³ and code examples illustrating the behaviors classified as most severe. Subsequently, we delve into the results from the semi-structured interviews with CS1 instructors, followed by an analysis of the frequency distribution of the most severe MC³. Finally, we conclude this section presenting the results obtained from the semi-structured observation conducted with CS1 instructors and students. For more detailed information on RQ1 and RQ2, we direct the reader to our Technical Report (Silva et al., 2023a).

4.1 MC³ Identification

As mentioned in Section 3, a total of six assignments were selected for analysis. The dataset consisted of 2,959 student submissions, out of which 2,874 passed all test cases. These submissions were the last ones that each student submitted for the assignments. Table 2 provides detailed information on the relevant topics covered in each assignment, along with the number of general submissions, correct submissions, and the subset of submissions that were analyzed. We chose to analyze roughly half (220) of the correct submissions for the last two assignments because they covered similar CS1 concepts (loops). Moreover, we believed that, by doing this, we would still have an adequate volume of material for an initial exploratory manual analysis. After applying these filters, we analyzed a total of 2,441 submissions.

After analyzing all 2,441 submissions, we identified a total of 45 MC³, which were split into eight distinct categories. Table 3 shows the list of MC³ with their severity classifications. The categories are named as follows: A) Variables, identifiers, and scope (A1 to A8); B) Boolean Expressions (B1 to B12); C) Iteration (C1 to C8); D) Function parameter use and scope (D1 to D4); E) Reasoning (E1 and E2); F) Test Cases (F1 and F2); G) Code Organization (G1 to G6); and H) Other (H1 to H3).

The topics listed in Table 2 correspond to the assignments appointed to students. Given the

Table 2: Description of how many student solutions to the assignments were submitted, correct (i.e. passed all test cases), and analyzed (i.e. checked by the researcher).

Related Topic	Submitted	Correct	Analyzed
Arithmetic operations: the **int** type	535	529	529
Arithmetic operations: the **float** type	499	491	491
Logical operations: the **bool** type	511	503	503
Conditionals I	499	478	478
Simple loops: **while**	459	452	220
Nested loops: **for**	456	421	220
Total	2,959	2,874	**2,441**

allotted time for submission (typically three weeks), students often learn about future concepts while working on prior assignments and may incorporate these concepts into their code. This phenomenon may account for the presence of MC³ related to concepts such as functions (category D) and lists (MC³ E2) in our dataset.

4.2 Questionnaire

A total of 32 volunteers participated in the questionnaire. The respondents were distributed across different countries as follows: Brazil (18), United States of America (9), Australia (1), Colombia (1), Finland (1), Slovenia (1), and The Netherlands (1). All answers received, including those with blank responses, were considered valid and included in the analysis.

Table 3 displays the MC³ severity ranking, presenting the ID, name, total number of responses for each Likert item category (strongly agree and agree (SA + A), neutral and blank (N + B), strongly disagree and disagree (SD + D)), and the calculated DIF (described in Section 3) for each MC³. The names given to the MC³ were carefully chosen to best describe the associated misconceptions. The table also includes a threshold indicated by a horizontal line, highlighting the most severe MC³. Any MC³ with a DIF value greater than 10 was classified as most severe, resulting in a total of 15 behaviors falling within this category.

In this study, we will concentrate our analysis on the 15 most severe MC³. To illustrate these misconceptions, we have created four sets of Python code. These code examples were constructed to provide generic samples of each coding behavior.

4.2.1 Set 1

This example, denoted in Code 3, contains 5 MC³: A4, D4, G4, G5, and H1. In line 9, the built-in function **max** was redefined (A4) by the user, creating it as a new function. In this same function, variables **a** and **b** were accessed, but they were not present in the function's scope (D4). Variables that were not significantly named (G4) were declared in the code, such as **a**, **b**, **c**, **x**, and **y**. The code was elaborated with an arbitrary organization (G5) as it alternated between input (line 1) and function declaration (line 2). This pattern was further repeated in lines 7, 8, and 9. Lastly, a statement with no effect (H1) was declared in line 4 because the result of the function **round** was not assigned to a variable.

Table 3: Severity ranking of the 45 identified MC³. Table is sorted decreasingly by the DIF column. The horizontal line highlights the 15 most severe MC³.

ID	Name	SA+A	N+B	SD+D	DIF
C8	**for** loop having its iteration variable overwritten	31	0	1	30
B6	Boolean comparison attempted with **while** loop	26	4	2	20
C1	**while** condition tested again inside its block	26	3	3	20
B8	Non utilization of **elif/else** statement	24	8	0	16
C2	Redundant or unnecessary loop	24	5	3	16
C4	Arbitrary number of **for** loop executions instead of **while**	24	5	3	16
D4	Function accessing variables from outer scope	24	4	4	16
G4	Functions/variables with non-significant name	24	7	1	16
H1	Statement with no effect	24	7	1	16
B12	Consecutive equal **if** statements with distinct operations in their blocks	23	5	4	14
B9	**elif/else** retesting already checked conditions	23	4	5	14
E2	Redundant or unnecessary use of lists	23	3	6	14
A4	Redefinition of built-in	22	3	7	12
F2	Specific verification for instances of open test cases	22	8	2	12
G5	Arbitrary organization of declarations	22	6	4	12
C3	Redundant operations inside loop	21	9	2	10
E1	Checking all possible combinations unnecessarily	21	7	4	10
G3	Too many declarations in a single line of code	21	7	4	10
A2	Variable assigned to itself	20	7	5	8
A6	Variables with arbitrary values (Magic Numbers) used in operations	20	6	6	8
A7	Arbitrary manipulations to modify declared variables	20	7	5	8
B11	Consecutive distinct **if** statements with the same operations in their blocks	20	6	6	8
B10	Unnecessary **elif/else**	19	9	4	6
B3	Arithmetic expression instead of Boolean	19	6	7	6
B4	Repeated commands inside **if-elif-else** blocks	19	11	2	6
D1	Inconsistent **return** declaration	19	6	7	6
A8	Arbitrary treatment of the stopping point of reading values	18	8	6	4
B7	Boolean validation variable instead of **elif/else**	18	5	9	4
C7	Arbitrary internal treatment of loop boundaries	17	6	9	2
C6	Multiple distinct loops that operates over the same iterable	16	9	7	0
F1	Verification for non explicit conditions	16	9	7	0
H2	Redundant typecast	16	8	8	0
G6	Functions not documented in the Docstring format	14	14	4	-4
A1	Unused variable	13	9	10	-6
A3	Variable unnecessarily initialized	12	8	12	-8
B1	Redundant or simplifiable Boolean comparison	12	12	8	-8
D2	Too many **return** declarations inside a function	12	8	12	-8
B5	Nested **if** statements instead of Boolean comparison	11	12	9	-10
G2	Exaggerated use of variables to assign expressions	11	13	8	-10
C5	Use of intermediary variable to loop control	10	11	11	-12
D3	Redundant or unnecessary **return** declaration	10	12	10	-12
H3	Unnecessary or redundant semicolon	8	8	16	-16
B2	Boolean comparison separated in intermediary variables	7	9	16	-18
G1	Long line commentary	7	8	17	-18
A5	Unused import	5	8	19	-22

4.2.2 Set 2

This example, denoted in Code 4, contains 4 MC³: B6, B8, B9, and B12. A **while** loop was used instead of an **if** (B6) to check if the sum of **num1** and **num2** was greater than 9 in line 4 because a **break** statement was declared in line 6. The non-utilization of **elif/else** (B8) in line 10 could have resulted in the value of **res** being overwritten (lines 9 and 11) depending on the value of **num2**. The **elif** declared in line 15 checked if the value of **num1** was not even. However, this check was unnecessary because it was already guaranteed when using an **elif** (B9). Lastly, the exact same condition was checked in lines 18 and 20, albeit with distinct operations inside each block (B12). These conditions could have been grouped in a single **if** statement.

```python
a = int(input())
def foo(a):
    a = a / 3.5
    round(a, 2)
    return a

b = int(input())
c = int(input())
def max():
    if b >= c:
        return b
    return c

x = foo(a)
y = max()
print(x, y)
```

Code 3: Examples of MC³: A4, D4, G4, G5, and H1.

```python
1   num1 = int(input())
2   num2 = int(input())
3
4   while num1 + num2 > 9:
5       print(num1 + num2, "has␣more␣than␣
            1␣digit")
6       break
7
8   if num2 <= 0:
9       res = num1 * num2
10  if num2 % 2 == 0:
11      res = num1 ** num2
12
13  if num1 % 2 == 0:
14      print(num1, "odd")
15  elif num2 % 2 == 0 and num1 % 2 != 0:
16      print(num2, "odd", num1, "even")
17
18  if num1 == num2 * 2:
19      print(num1, "multiple␣of", num2)
20  if num1 == num2 * 2:
21      print(num1, "odd")
```

Code 4: Examples of MC³: B6, B8, B9, and B12.

4.2.3 Set 3

This example, denoted in Code 5, contains 4 MC³: C1, C2, C4, and C8. A **while** loop, declared in line 16, had its condition verified again in its block (C1) in line 19. There was no need to verify **numMax** again since it was set at the end of the loop. The **for** loop declared in line 9 was executed only once (C2), thus making it unnecessary. In line 2, another **for** loop was declared to read and add values to a list. However, it was arbitrarily declared (C4) to be executed 9999 times, hoping that the stopping condition (line 4) would happen before reaching the maximum iteration value. Lastly, the **for** loop declared in line 12 had its iteration variable **k** overwritten (C8) inside the loop's body, in line 14.

4.2.4 Set 4

This example, denoted in Code 6, contains 2 MC³: E2 and F2. A list was used to store input values in lines 2 to 5. However, the storing process was not necessary (E2) if the purpose was only to sum these input values, as described in the declared loop in line 8. In this case, **totalSum** could have been calculated while reading the input. Now, suppose that the set of input from open test cases was $I = \{\{1,1,1\},\{2,2,2\},\{1,2,3,4,5\}\}$ and the expected output was $O = \{\{3\},\{6\},\{15\}\}$. To obtain the correct result, the code did not use the value of **totalSum**, but rather printed the expected values for each specified entry (F2) in lines 11, 13, and 15.

```
1   numList = []
2   for i in range(9999):
3       a = int(input())
4       if a == 0:
5           break
6       numList.append(a)
7
8   numMax = max(numList)
9   for j in range(1):
10      print(numMax)
11
12  for k in range(numMax):
13      print(k + 1)
14      k += 2
15
16  while numMax != 0:
17      print(numMax)
18      numMax = numMax - 1
19      if numMax == 0:
20          break
```

Code 5: Examples of MC³: C1, C2, C4, and C8.

```
1   numList = []
2   num = int(input())
3   while num != 0:
4       numList.append(num)
5       num = int(input())
6
7   totalSum = 0
8   for item in numList:
9       totalSum += item
10
11  if numList == [1, 1, 1]:
12      print(3)
13  elif numList == [2, 2, 2]:
14      print(6)
15  elif numList == [1, 2, 3, 4, 5]:
16      print(15)
```

Code 6: Examples of MC³: E2 and F2.

4.3 Interviews with CS1 Instructors

Out of the 32 instructors who had answered the questionnaire, 18 volunteered to participate in the semi-structured interview. We were able to conduct the interviews with nine participants: seven from Brazil and two from the United States of America. The average interview duration was approximately 42 minutes.

All interviewees stated that they have been using Python in their CS1 courses recently. Instructors mentioned assigning summative tasks to students, which varied from lists of exercises to coding projects. Six participants confirmed using autograders to assess these tasks, while three preferred manual assessment, strictly not using autograders.

The way in which instructors use autograders also varies. Regarding students' submissions, three instructors stated that they manually check them, even if the code passes the test cases. One instructor mentioned that he does not check the submissions, while two instructors stated that they only check if the assignment is complex. Among the respondents who do not use these systems, one mentioned that he is not familiar with them, and two others stated that, since their classes are small, they prefer to manually evaluate the assignments. They argued that this is the best way to assess if students are understanding the concepts taught.

Interviewees mentioned that they have encountered the following MC³: B1, C2, F2, and G4. Instructors also identified similar behaviors, such as the inadequate use of functions (e.g., using a function that encapsulates the entire code), using lists when other data structures would have been more appropriate, and inconsistent code style (e.g., spacing between lines and characters).

All instructors expressed their interest in an autograder that can detect MC³. However, they also agreed that the feedback provided to students should be configurable, as five interviewees emphasized that too much information can have a negative impact. Other suggested features for this autograder included the application of machine learning techniques to teach code refactoring, evaluation of code complexity, relaxing the strictness of automated correction (removing the binary factor if a test case was passed or failed), a dashboard showing statistics on the occurrence of MC^3, and means to verify if students are implementing the feedback related to MC^3 in their code (such as questionnaires).

As an additional intervention method for addressing MC^3, eight instructors expressed interest in using Peer Instruction (PI) (Crouch & Mazur, 2001). PI is an Active Learning technique that promotes meaningful discussions among students regarding different comprehension aspects of a topic. In the context of MC^3, we proposed an idea where, after introducing a new CS1 topic in class, instructors would administer a multiple-choice question. This question would present code snippets constructed with MC^3, and students would attempt to identify and understand why these coding behaviors should be avoided. However, the interviewed instructors highlighted that the optimal time to use this technique would be after students have become familiar with the relevant CS1 topics. Additionally, interviewees expressed concerns about the timing of administering these questions. They suggested that it would be best to incorporate them in a dedicated class session, such as one focusing on code quality. Only one instructor expressed disinterest in using this technique, citing limited time to implement it with undergraduate students. The instructor also raised concerns about the potential for cheating or guessing among students when answering multiple-choice questions.

4.4 MC^3 Frequency Distribution

As mentioned in Section 3, we conducted an analysis of all MC102 assignments for the first term of 2020 and the second term of 2022. The first term consisted of 14 assignments, while the second term consisted of 15. Since the assignments in both terms explored similar CS1 concepts, we grouped them together based on their general topics. This grouping resulted in the identification of seven distinct main topics, encompassing a total of 11,141 correct submissions. The distribution of submissions per topic is presented with the MC^3 frequency in Table 4.

By using the Python AST module, we managed to implement an automatic detection for 14 out of the 15 MC^3 classified as most severe. To simplify the implementation, we focused on the fact that the analyzed code would be somewhat simple because it would come from CS1 assignments solved by CS1 students. This allowed us to strive for a generic code for automated detection as we could check simple cases that comprised the MC^3. For instance, regarding the MC^3 C1 (**while** condition tested again inside its block), our detection only checks for **while** loops in which the condition is composed of only one variable. Our rationale for this was because it is rare for CS1 students to declare a **while** loop with a condition comprised of more than one variable. Another example is the detection for the MC^3 B8 (Non utilization of **if-elif-else**). We limited it only to check if there was declared an **if-elif** structure (without an **else**). In this case, our rationale was to point out to the students that, probably, either their last **elif** could have been an **else**, or the whole decision structure could have been made by only **if**s, as they were already mutually exclusive. In light of this approach, we were not able to create a generic automated detection for the MC^3 F2 (Specific verification for instances of open test cases) since it depended heavily on the test cases.

Another challenge we faced was addressing the MC³ C4 (Arbitrary number of **for** loop executions instead of **while**), E2 (Redundant or unnecessary use of lists), and G4 (Functions/variables with non-significant names). We recognized that these asserting these MC³ depends on the instructor's perspective. To address this, we introduced thresholds that can be set by the instructor to determine the presence of these MC³ in the code. For C4, the code determines the presence of the MC³ if the number of iterations in the **range**-based **for** loop is greater than or equal to a constant (set as 7 in our results). Similarly, for E2, we count the number of lists in the code and check if it exceeds a constant threshold (set as 10 in our results). As for G4, we established three constants: the minimum number of characters for variables' names, the minimum number of characters for functions' names, and the percentage threshold of variables or functions that can have fewer characters than the specified thresholds. If the percentage of variables or functions outside the specified thresholds exceeds the given percentage threshold, the code is considered to have the MC³. In our results, we set the minimum thresholds to 4 characters for variables, 8 characters for functions, and 70% as the percentage threshold. We defined these constants based on a small subset of students' submissions we checked manually for assignments in both academic terms.

Table 4 presents the distribution of the analyzed correct submissions for each topic as well as the MC³ frequency. The DIF values are the same presented in Table 3. The number of correct submissions per topic varied based on the number of assignments for each term. Since the number of submissions varied for each topic, we calculated the frequency as a percentage of correct submissions in which the respective MC³ was exhibited. Since a single submission could have more than one distinct MC³, the columns might not add to 100%. In Section 5, we delve into the reasons behind these variations and discuss how they impact our findings.

Table 4: Frequency distribution of the most severe MC³ organized by assignment topics. The analyzed correct submissions is presented in parenthesis for each topic. The frequency is a percentage of correct submissions that exhibited the respective MC³. Table is sorted decreasingly by Total.

MC³	DIF	Types, I/O, Operations (2,662)	Conditional Commands (1,445)	Repetition Commands (1,855)	Lists, Tuples, Strings, Dictionaries (2,928)	Matrices (1,237)	Searching and Sorting Algorithms (813)	Recursion (201)	Total (11,141)
G4	16	61.8	44.8	26.6	19.1	32.7	26.3	60.7	36.7
B8	16	0.5	55.7	32.6	27.7	45.6	20.4	13.4	26.8
G5	12	0.1	1.0	1.6	5.5	30.0	29.2	32.3	7.9
D4	16	0.0	0.6	0.5	3.3	30.8	17.1	39.8	6.4
B9	14	0.0	3.2	5.5	2.8	0.2	3.9	0.0	2.4
A4	12	0.3	2.8	2.3	1.7	2.1	4.6	3.5	1.9
C8	30	0.0	0.1	2.5	2.8	2.7	4.7	1.0	1.8
C1	20	0.0	0.0	1.0	2.0	0.1	2.1	0.5	0.9
E2	14	0.0	0.1	0.2	0.4	3.3	0.9	2.0	0.6
B12	14	0.0	1.7	0.2	0.0	0.1	0.0	1.0	0.3
H1	16	0.3	0.3	0.3	0.2	0.3	0.1	0.0	0.3
C4	16	0.0	0.1	0.3	0.2	0.3	0.4	1.0	0.2
C2	16	0.0	0.1	0.2	0.3	0.6	0.0	0.0	0.2
B6	20	0.0	0.1	0.2	0.0	0.0	0.0	0.0	0.0

4.5 Observation in a CS1 Course

The semi-structured observation took place in one MC102 class during the second term of 2022 and another class in the first term of 2023. In 2022, a researcher observed the lectures and had conversations with students. In 2023, the same researcher conducted only the conversations. In total, 20 students and one instructor participated in the activities. While the researcher observed and analyzed all lectures and educational materials, he evaluated all assignments except for the ones regarding Searching and Sorting Algorithms and Recursion (following the topics in Table 4).

4.5.1 Conversations with Students

Based on the researcher's notes, we identified students' reasons for 10 MC³: A4, B8, B9, C1, C8, D4, E2, G4, G5, and H1. In general, two recurring comments were observed regarding these MC³. Firstly, students expressed a concern about ensuring that their code passed all test cases. This led to the development of MC³ such as C1 and B9, where students included redundant checks to guarantee code correctness. This same mindset also influenced other behaviors, including D4, G4, and G5, as students, focused only in the functioning of their code, neglected coding guidelines such as code organization, providing all function arguments, and using meaningful variable names. Secondly, students also expressed to have a careless approach to coding. MC³ A4 emerged due to students unintentionally redefining built-in functions without realizing it. Additionally, instances of B8 occurred when students left unused code snippets in their solutions or just used a copy-paste approach with **elif** statements, neglecting to consider an **else** clause.

We also observed instances where students demonstrated incomplete comprehension of CS1 topics, leading to the occurrence of MC³. For example, instances of C1 arose because students believed that the **while** loop condition must be checked at the end of its body. Similarly, in the case of C8, students mistakenly thought that the iteration variable of a **for** loop needed to be manually updated, leading them to overwrite it. Another common observation was students coding with B9 due to a lack of understanding of how **elif** statements work, resulting in the repetition of already checked conditions. Furthermore, students, struggling with the concept of decision structures, resorted to coding with only **if** statements, leaving their code prone to errors. Lastly, this misunderstanding of decision structures also influenced the occurrence of H1. Students mistakenly believed that **else** clauses were mandatory and, without knowing what to express in them, included statements with no effect to not alter the code output.

Lastly, students expressed their preferences for specific coding practices. One reason for the occurrence of E2 was that students felt comfortable deliberately using lists. This behavior stemmed from either a lack of awareness that lists were unnecessary for the task or their familiarity with certain features of lists (e.g., using the **sum()** built-in function). Similarly, the comfort students felt while coding influenced the occurrence of G4, as they preferred using simple letters as variable names for the sake of ease and speed. We also observed that the nature of the assignments themselves had an impact on the occurrence of E2 and G4. For example, as the specific assignment for dictionaries did not force the use of this data structure, students resorted to solutions created solely using lists. Likewise, when assignments named entry values with non-significant names, students adopted these names in their code and created other variables with similar non-significant names based on the instructions provided.

4.5.2 Assessment of Educational Materials

The researcher only identified the MC³ A4 while analyzing the lecture slides and code snippets developed in the classes. A code sample in the slides for Strings redefined the built-in type **str** by assigning it to a new variable. The same behavior happened in the slides for Dictionaries, albeit it was the **max()** function that was redefined as a variable. The last built-in function that was redefined was **bin()**. This one was set as the argument of a user-defined function in the slides for Recursion. In this case, there was a redefinition even if it only pertained to the function's scope. However, students who developed the MC³ in their code did not justify having seen it in class.

We presented our findings to the instructor at the end of the 2022 term. In general, he said he had overlooked the scenarios where a redefinition of built-ins can lead to because A4 was present in code snippets that served a simple purpose in the slides. We presented the concepts behind our study of MC³, and the instructor expressed interest in learning more. He also suggested exploring automated detection using machine learning techniques. The slides that contained the MC³ A4 had the code snippets modified for the next term in 2023.

5 Discussion

In relation to RQ1 and RQ2, although we had initially aimed for a higher number of volunteers, we found the geographic distribution of the 32 respondents to the questionnaire to be satisfactory. Furthermore, all nine interviewees represented different institutions. Therefore, we conclude that these factors contributed to gathering opinions from CS1 instructors teaching in diverse contexts, allowing us to assess the severity of and potential approaches to addressing MC³ in CS1 classes.

As for RQ3 and RQ4, although both analyses were conducted within a single institution, we examined students' submissions for 19 different bachelors' programs MC102 in 2020 and 20 in 2023 to calculate the frequency distribution of MC³. Additionally, the participants in the semi-structured observation represented different programs, namely bachelor's programs in chemistry engineering, statistics, and agricultural engineering. Given the variation in programs across the terms analyzed, we can conclude that our analysis encompassed students with diverse backgrounds and objectives in both research questions.

5.1 MC³ Severity and Reasons for Occurrence (RQ1 and RQ4)

The identification and assessment of the MC³ followed some of the guidelines described by Almstrum et al. (2006). Although our aim was not to develop a CI, we analyzed open-ended MC102 assignments, consulted experts in the field (survey with CS1 instructors), and conducted observations to understand the process by which students develop misunderstandings (conversations with MC102 students). By conducting these three assessments, we were able to mitigate potential biases in the opinions of the researchers who identified the MC³. In the following subsections, we discuss the most severe MC³ within each category (highlighted in Table 3), incorporating the opinions collected from CS1 instructors and explaining why MC102 students incorporated those MC³ into their code.

5.1.1 A: Variables, identifiers, and scope

In this category, only A4 was classified as most severe. Instructors stated that the severity is associated because this behavior can lead to unexpected errors in the code that will be difficult to detect. However, instructors also stated that the occurrence is intrinsically related to Python and the natural language students choose to program (e.g., English or Portuguese). On the other hand, students explained that they did not pay attention while programming, thus incorporated this MC³ in their code. In our analysis, we identified redefinitions of built-in types (e.g., **bool**, **dict**, **list**, and **str**) and of built-in functions (e.g., **min()**, **max()**, **len()**, **input()**, among others). Almost all those redefinitions happened when students created variables with those names in their code. We agree with the opinions stated by CS1 instructors, particularly when students are redefining built-in types because typecasting is a common in CS1 assignments, especially when reading data from external sources. While we also identified this category present in other work (De Ruvo et al., 2018; Keuning et al., 2021; Oliveira et al., 2023), this behavior did not appear in them.

The occurrence of A4 depends on the natural language used by the student for programming. Students who prefer using English-based variable or function names are more likely to redefine built-ins. While it is not possible to redefine built-in types in strongly typed languages, it is possible to redefine built-in methods in languages like Java. However, we argue that this is less likely to occur arbitrarily, as opposed to built-in functions in Python.

5.1.2 B: Boolean expressions

MC³ B6, B8, B9, and B12 were classified as the most severe in this category. Instructors generally agreed that these behaviors indicate a lack of clarity in students' thought processes when developing their own code. This lack of clarity often stems from an incomplete or poor comprehension of decision statements. Another possible reason could be the development of poor coding habits resulting from previous attempts by students to learn programming.

For MC³ B6 and B12, we did not obtain specific results regarding why students develop these misconceptions. However, the incorrect use of the **while** statement in B6 may be related to students attempting to apply a newly learned concept in their code, even when it is not necessary. This phenomenon is referred to as *knee-jerk* (Ureel II & Wallace, 2019). Soloway and Ehrlich (1984) describe MC³ B6 as follows: "An IF should be used when a statement body is guaranteed to be executed only once, and a WHILE used when a statement body may need to be repeatedly executed" (p. 597). As for MC³ B12, one possible reason is that students are attempting to emphasize both conditional blocks with the same **if** statement. In this case, students may or may not be aware that these blocks could have been merged. RPT (Keuning et al., 2021) has rules to extract duplicate declarations inside **if/else** statements, and Oliveira et al. (2023) identified misconceptions when students try to refactor these declarations, such as keeping unnecessary **else** blocks and incorrectly updating necessary Boolean expressions.

Students provided different reasons for exhibiting MC³ B8. Among those who mentioned being absent-minded while programming, it was either because they perceived their code to be already correct or because they wanted to quickly complete the assignment. However, other students mentioned being unfamiliar with **elif** or **else** statements, resulting in their avoidance of using these constructs in their code. This aligns with what the instructors mentioned about incomplete or poor comprehension of decision structures. In our opinion, students may simply be distracted and mistakenly use **elif** instead of an appropriate **else** statement. However, coding with only **if** statements can lead to buggy programs, as the stated conditions may not be mutually exclusive. This latter behavior is indeed more severe and should be addressed accordingly.

The reasons behind students coding with MC³ B9 were similar to those for B8. In addition to students attributing it to being absent-minded while programming, we also identified comments indicating a poor comprehension of the **elif** statement. Although one instructor mentioned that students exhibiting this behavior may do so to help organize their thought process, we consider that if this misconception persists in later parts of a CS1 course, it indicates a fundamental misunderstanding. Table 4 illustrates that B9 did indeed appear throughout the analyzed assignments. De Ruvo et al. (2018) identified MC³ B9 as 'Unnecessary IF/ELSE'.

When considering that MC³ B6, B8, B9, and B12 all pertain to conditional statements, we argue that these MC³ can occur in programming languages that employ such constructs. While the use of the **elif** statement is exclusive to Python, all the aforementioned MC³ can also manifest in other programming languages using only **if-else** statements.

5.1.3 C: Iteration

Among the MC³ in this category, C1, C2, C4, and C8 were considered the most severe. According to the instructors, these behaviors primarily stem from students' lack of comfort with a particular structure, leading them to prefer one over another. One common commentary was that instructors intentionally do not teach the **break** statement to discourage coding behaviors associated with these MC³. The use of **break** is a topic of discussion among both CS1 instructors and the programming community at large (Sorva & Vihavainen, 2016).

We were unable to gather specific information on why students implemented MC³ C2 and C4 in their code. In the case of C4, we agree with the instructors' views that students' preference for a specific construct indicates a lack of understanding. Instead of using a **while** loop, students with this behavior replace it with a **for** loop with an arbitrary number of iterations. This suggests a potential misunderstanding of **while** loops. On the other hand, we argue that C2 is another example of the knee-jerk phenomenon. In this case, students use loops that execute only once because they have recently learned the concept and think it is necessary to apply it.

Regarding the MC³ C1, students provided two main reasons for their behavior: either they wanted to ensure the correctness of their code, or they were unaware that manually checking the **while** condition was unnecessary. In both cases, evidence suggests that students have an incomplete understanding of how the **while** construct works, and this misconception should be addressed when identified. RPT has refactoring rules similar to C1 and other previously mentioned MC³, such as removing **break** statements in a loop or rewriting a **for** loop with a **while** (Keuning et al., 2021). Oliveira et al. (2023) also identified misconceptions when students attempt these refactorings, such as replacing a **for** loop with an incorrect **while** or **for-each** loop.

In the case of the MC³ C8, students stated that they either believed incrementing the iteration variable was necessary or alleged overwriting the said variable due to inattention. These commentaries exemplify an incomplete understanding of loop constructs, as students are confusing **while** and **for** loops. Additionally, the inattention in overwriting the iteration variable can be attributed to students' lack of experience. We agree with CS1 instructors who emphasized the need to address this behavior to prevent code malfunctions that may be difficult to detect and correct in the future. Similar to the previous category, MC³ C1, C2, C4, and C8 can manifest in other programming languages with iteration constructs. While there may be some variations, such as with the **for-loop** in Java, we argue that these misconceptions can generally be replicated.

5.1.4 D: Function parameter use and scope

In this category, only the MC³ D4 was considered as most severe. CS1 instructors stated that although this is another Python specific misconception, it can lead to bad coding practices in the future, such as code that is both difficult to read and maintain. Instructors also noted that this behavior is often observed in students with previous coding experience. Based on the students' explanations, it was identified that D4 was primarily caused by inattention while programming. In this case, students did not pass variables as arguments to a declared function but observed that it did not affect the correctness of the code. Therefore, students did not bother passing variables as arguments. To promote the practice of writing readable and maintainable code, we argue that this MC³ should be properly addressed when detected.

The occurrence of D4 is related to the language being static or dynamic typed. In Python, this MC³ can occur more readily since variables do not need to be explicitly declared as globals to be used inside functions without being passed as parameters. The use of globals is discouraged in CS1 and D4 should not happen in any programming languages.

5.1.5 E: Reasoning

MC³ E2 was the only one classified as most severe in this category. According to CS1 instructors, they only see a problem with lists that are created and used only once, suggesting that students who exhibit this behavior may have misunderstood the concepts of lists. Instructors also mentioned that this MC³ is common and difficult to rectify. Based on students' comments, the predominant reason for using lists in this manner is the comfort associated with this structure. Students also mentioned using lists to store input first and then consume it later.

While it is acknowledged that students may be organizing their thoughts by creating composition plans (Fisler et al., 2016; Soloway & Ehrlich, 1984) that involve separating input from consumption, our analysis suggests that some students may develop a reliance on lists as the default solution for any assignment. During conversations about the assignment designed for the use of dictionaries in Python, students who did not use this data structure in their solutions claimed to have managed using only lists. Although the use of dictionaries was not mandatory in the assignment, these students ended up with larger and more complex code. Based on this observation, we reason that a clear and meaningful use of lists should be addressed in CS1 to mitigate this MC³. We argue that the occurrence of E2 is related to the ease of use of lists in Python. Students might not develop this MC³ in C, for example, but it is possible in other languages such as C++ or Java.

5.1.6 *F: Test cases*

Among the MC³ in this category, only F2 was classified as most severe. According to instructors, students exhibiting this behavior may either struggle with understanding the functionalities of the autograder or attempt to cheat the system. However, during the conversations with students, we did not specifically identify this particular MC³. Our conclusion is based on the instructors' opinions, suggesting that students displaying this behavior may have difficulties to understand the concepts of input and output. The occurrence of F2 is solely dependent on the use of autograders.

5.1.7 *G: Code organization*

The most severe MC³ in this category were G4 and G5. Instructors emphasized that variables or functions with insignificant names should only be used in coding drafts, not in the final submissions. They also commented that disorganized code reflects a lack of clarity in students' reasoning, which explains why students arbitrarily define functions. Instructors generally agreed that both MC³ should be addressed early in CS1 courses to promote code legibility and maintainability.

Regarding G4, students provided various explanations. Students mentioned that they used small, alphabetical variable names to quickly develop their code. Others claimed to be influenced by the assignment's description. For instance, if in the description was said that variables **a**, **b**, and **c** were specified for the sides of a triangle, students naturally created these variables in their code. While these names were meaningful in the context of the assignment, our analysis revealed that students also used arbitrary alphabetical letters for other variables required for features such as triangle classification. Based on this, we underscore the importance of teaching significant naming in CS1 classes while ensuring that assignment descriptions align with this principle.

As for G5, students echoed the reasons mentioned by the instructors. They stated that they created functions during their thought process while solving the assignment, and since their code was correct, they did not prioritize organizing them. We argue that this behavior arises because students are not being assessed on code organization. However, it is crucial to address this issue when detected to code readability and maintainability. Since naming variables or functions and the organization of declarations is commonplace in other programming languages, we also argue that the occurrence of both G4 and G5 are not exclusively to Python.

5.1.8 *H: Other*

Among the MC³ in this category, only H1 was classified as the most severe. Instructors stated that this behavior often arises from a lack of attention while students are coding and, if left unaddressed, can result in future bugs. While it is common at the beginning of the course, instructors mentioned that if it persists in later parts, it indicates that the student is struggling with the underlying concepts. In our conversations with students, we observed instances of loose declarations, such as a **True** statement, placed within the body of an unnecessary **else** clause. Students explained that they believed the loose statement was necessary to maintain the code's functionality. While we acknowledge that this MC³ is related to a lack of attention while coding, it can be mitigated by encouraging students to refine their code even after it is correct. De Ruvo et al. (2018) listed a semantic style similar to H1 called "Useless Declaration / Assignment Division". We are unaware of anything that can impede the occurrence of H1 in other programming languages.

5.2 Addressing MC³ in CS1 Classes (RQ2 and RQ3)

The decision to study MC³ in Python was well-founded as this language is widely utilized (Becker & Fitzpatrick, 2019; Guo, 2014). This assertion was further supported by the CS1 instructors who were interviewed. Our findings revealed that CS1 students are assigned various types of programming tasks, ranging from practice exercises to final projects. MC³ can emerge within the process of students' developing their solutions to these multiple types of assignments. This factor underscores the need for diverse approaches to address MC³ within CS1 classes.

5.2.1 Insights from CS1 Instructors

We observed that the decision to use autograders in CS1 classes depends on various factors, such as class size and the complexity of configuring the autograder for different assignments. Considering these factors, an effective autograder capable of detecting MC³ would need to be adaptable and applicable in diverse teaching contexts of CS1. Furthermore, according to the instructors, automated detection alone is not sufficient. The feedback provided by the tool is a crucial aspect that can either motivate or demotivate students. Henceforth, we argue that careful consideration should be given to the construction and delivery of feedback, as overly technical information may not be helpful, especially for beginners.

We also noted that implementing PI would require preparation and adjustment of CS1 courses, as half of the interviewees admitted being unfamiliar with Active Learning techniques. This implementation would also increase the workload for instructors (Caceffo et al., 2019). However, despite these challenges, research has demonstrated positive outcomes when employing Active Learning techniques (Simon, Esper, et al., 2013; Simon et al., 2010). For example, students achieved better results compared to traditional teaching methods (Simon, Parris, & Spacco, 2013), and failure rates were reduced (Porter et al., 2013). These results serve as motivation for the use of PI in CS1 classes. We argue that integrating PI with the study of MC³ would not only complement the feedback provided by an autograder capable of detecting these behaviors but would also provide new insights into why students code with MC³.

5.2.2 Insights from MC102 Students

After concluding the conversations with students, we distributed a survey to gather additional insights about their experiences with learning about MC³. Out of the 20 participants, eight responded to the questionnaire. Although the response rate was relatively low, these responses provided valuable insights into how students reflected on the MC³ discussed during the conversations and how they believe MC³ could be addressed in the context of MC102.

Students shared that they have become more attentive to code organization, emphasizing the importance of using meaningful variable names (MC³ G4) and declaring functions at the beginning of their code (MC³ G5). In terms of Boolean expressions, students mentioned that they have learned about the functionality of the **elif** statement, thus avoiding checking already performed checks in previous **if** statements (MC³ B9). Additionally, students reported that their misunderstandings regarding **if-elif-else** constructs have been clarified. Lastly, students expressed increased awareness to avoid rechecking the **while** condition at the end of its body (MC³ C1).

Overall, students expressed agreement that MC³ should be addressed in the classroom. They provided several suggestions on how to incorporate discussions about MC³ into the teaching process. One suggestion was for the instructor to select anonymized student solutions to the assignments and discuss the MC³ present in these solutions during lectures. If not done during lectures, students suggested that teaching assistants could fulfill this role during practice hours. Another suggestion involved sending individual e-mails to students, explaining the specific MC³ found in their assignment solutions. One student emphasized the importance of timely feedback on MC³, as they would not remember their solutions after some time had passed between submission and classroom discussions. Lastly, a student proposed that the class slides for each CS1 topic should already include information about the most frequent MC³.

The feedback received from students indicated that they found the conversations to be positive and helpful. Students' suggestions for addressing MC³ in CS1 classes complemented insights from interviewed instructors. These suggestions and insights were crucial to understand the reasons for the development of MC³ and should provide a foundation to the design of formative feedback messages, which can later be incorporated to autograders.

5.2.3 Insights from MC³ Automated Detection

The frequency distribution of MC³, as presented in Table 4, reveals that these misconceptions can occur throughout the entire duration of a CS1 course, although their occurrences are generally not high. It is important to note that, on a first glance, certain MC³ should not be prone to occur in assignments conducted before the corresponding CS1 topic is taught. For instance, in MC102, repetition commands are taught after conditional commands. One might assume that students would not develop MC³ related to loops before the concept of loops is introduced. However, the table indicates occurrences of C8, C2, and C4 in assignments focused on conditional commands. Similarly, MC³ D4 and G5, which are function-related misconceptions, occur before functions are taught in the same CS1 course, which is right before the topic of matrices. Our evidence suggests that students developing misconceptions before the teaching of the related CS1 topic may indicate prior programming knowledge, which aligns with the observations made by CS1 instructors.

The distribution of MC³ G4 and B8 should be interpreted with caution. As mentioned earlier, our initial development of the automated detection system relied on generic rules as a foundation to keep its usage simple. In the case of G4, we used constant thresholds to determine whether variables and functions had significant names based on their character length. This approach was influenced by the variable and function names provided in the assignment descriptions, as students often used these names in their code. Consequently, some names that met our thresholds might still be considered insignificant based on the assignment context. Regarding B8, our detection was limited to identifying **elif** statements without an accompanying **else** statement. While we acknowledge that this issue should be addressed in feedback, we recognize that it is not as severe as the presence of consecutive **if** statements that are not mutually exclusive.

We acknowledge that linting tools, which identify bugs, errors, and code anomalies according to a specific coding style (De Ruvo et al., 2018), could also detect MC³, particularly those in category **A**. However, these tools often provide extensive feedback that may overwhelm CS1 students (Keuning et al., 2019, 2021). Although their feedback can be customized, it raises the same regarding instructors' increased workload. Furthermore, the effectiveness of these tools depends on the topics covered in each course, as the CS1 syllabus may vary (Becker & Fitzpatrick, 2019; Silva et al., 2023c). Evidence suggests further research to explore and develop viable and comprehensive approaches to integrating automated MC³ detection in CS1 courses.

6 Limitations and Threats to Validity

The primary limitation of this study regards the teaching context of the CS1 course used for identifying and analyzing the MC³. Although this course was from a single institution, the analysis was composed of different undergraduate programs. Additionally, the course taught the Python programming language using the imperative paradigm. We recognize that MC³ may occur in other languages, but some misconceptions like A4, D4, and E2 are more prone to happen in Python. Therefore, we consider that replication of this research may differ when conducted in CS1 courses that use different programming languages and paradigms.

Another limitation stems from the dataset used to identify MC³. While the analyzed assignments (Table 2) and the identified MC³ categories (Table 3) represented topics listed as most covered in typical CS1 courses (Becker & Fitzpatrick, 2019; Silva et al., 2023c), we acknowledge that our MC³ list may not be exhaustive. Specific MC³ related to other CS1 concepts may exist. However, since our data demonstrated students incorporating MC³ throughout the entire semester, we argue that the identified listing remains significant for addressing MC³ in CS1 classes.

The subjective nature of identifying the MC³ poses the main threat to the validity of this study. To mitigate this concern, we developed the survey with CS1 instructors. The respondents' geographic distribution, along with their diverse teaching contexts, helped identify the most severe MC³ that should be adequately addressed in CS1 classes.

7 Conclusions

The objective of this study was to identify characteristics in code, which, despite passing all test cases in an autograder, indicated faulty or incomplete understandings of CS1 concepts. These identified characteristics were termed Misconceptions in Correct Code (MC³). By analyzing 2,441 student submissions to assignments in a CS1 course taught in Python using the imperative paradigm, we identified a total of 45 MC³. These misconceptions were divided into eight categories: A) Variables, identifiers, and scope; B) Boolean expressions; C) Iteration; D) Function parameter use and scope; E) Reasoning; F) Test cases; G) Code organization; and H) Other.

To determine the severity of each MC³ and prioritize those that required immediate attention in the classroom, we conducted a survey with CS1 instructors. A total of 32 instructors participated in an online questionnaire, which included Likert-item questions to assess the severity of each MC³. Additionally, nine of these instructors took part in a semi-structured interview, aimed at exploring different strategies to address MC³ in various teaching and learning contexts within CS1 courses. Furthermore, to gain insights into the reasons why students incorporated MC³ in their code, we conducted a semi-structured observation in a CS1 course. This observation involved 20 students and one instructor. Additionally, we developed an automated detection method, based on static code analysis, for the MC³ identified as the most severe. All methods that included research with human participants were first assessed and approved by an Ethics Research Committee.

We have identified and ranked 15 MC³ as the most severe out of the total identified misconceptions. Among these, eight are directly related to core concepts of Boolean expressions and iteration, which are fundamental in a typical imperative-based CS1 course. Both instructors and students provided insights into the reasons behind the development of these MC³. Some of these reasons included: students' misconceptions about Python constructs, such as decision statements and loops; as well as a careless approach to code development, where the focus is solely on achieving correctness with regard to test cases. Instructors and students have suggested various strategies to address MC³ in CS1 classes: integrating the detection of these misconceptions into an autograder to provide formative feedback; incorporating MC³ into lecture slides or during practice hours; and integrating them into Active Learning techniques like Peer Instruction. Additionally, our initial implementation of automated detection revealed that while these misconceptions may not occur in large numbers, they are distributed throughout the entire CS1 course.

Based on the evidence obtained, it is concluded that research on MC³ is well-founded as it provides valuable assistance to CS1 students and instructors by identifying underlying misconceptions that can persist even in correct code. If left unaddressed, these misconceptions may carry over into subsequent CS courses, hindering students' progress. The 15 MC³ identified as the most severe were found to be rooted in faulty understandings of CS1 topics and a lack of attention to coding characteristics such as readability and maintainability. While there is ongoing discussion among instructors and teaching assistants regarding the prioritization of factors like code efficiency versus readability and maintainability (Barbosa et al., 2023; Fisler et al., 2016), researchers argue that stimulating behaviors related to the latter is more beneficial for CS1 students (De Ruvo et al., 2018; Joni & Soloway, 1986; Keuning et al., 2019). Therefore, we advocate that addressing MC³ in CS1 classes should involve educational materials taught by both instructors and teaching assistants, assignment design, and formative feedback. The insights obtained by students and instructors regarding the reasons behind MC³ occurrence serve as basis for creating feedback messages that can improve teaching and learning outcomes (Cain & Babar, 2016). The aim is to provide timely and formative feedback that closely aligns with the guidance an instructor would offer since the purpose of the course is to teach and not to grade (Edwards, 2021).

Acknowledgments

We would like to thank the 32 respondents to the online questionnaire, the nine instructors that participated in interviews, and the 20 students and the instructor that participated in the observation. We would also like to thank the Brazilian National Council for Scientific and Technological Development (CNPq) for the financial support (process numbers 142476/2020-0 and 312088/2020-5), as well as the São Paulo Research Foundation (FAPESP) for grant number 2013/08293-7.

Extended Awarded Article

This publication is an extended version of an awarded paper at the III Brazilian Symposium on Computing Education (EduComp 2023), entitled "Passar nos casos de teste é suficiente? Identificação e análise de problemas de compreensão em códigos corretos" (Silva et al., 2023b). DOI: 10.5753/educomp.2023.228346.

References

Ali, M., Ghosh, S., Rao, P., Dhegaskar, R., Jawort, S., Medler, A., Shi, M., & Dasgupta, S. (2023). Taking stock of concept inventories in computing education: A systematic literature review [GS Search], 397–415. https://doi.org/10.1145/3568813.3600120

Almstrum, V. L., Henderson, P. B., Harvey, V., Heeren, C., Marion, W., Riedesel, C., Soh, L.-K., & Tew, A. E. (2006). Concept inventories in computer science for the topic discrete mathematics [GS Search]. *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, 132–145. https://doi.org/10.1145/1189215.1189182

Araujo, A., Filho, D., Oliveira, E., Carvalho, L., Pereira, F., & Oliveira, D. (2021). Mapeamento e análise empírica de misconceptions comuns em avaliações de introdução à programação [GS Search]. *Anais do Simpósio Brasileiro de Educação em Computação*, 123–131. https://doi.org/10.5753/educomp.2021.14478

Araujo, L. G., Bittencourt, R., & Chavez, C. (2021). Python enhanced error feedback: Uma ide online de apoio ao processo de ensino-aprendizagem em programação [GS Search]. *Anais do Simpósio Brasileiro de Educação em Computação*, 326–333. https://doi.org/10.5753/educomp.2021.14500

Austing, R. H., Barnes, B. H., Bonnette, D. T., Engel, G. L., & Stokes, G. (1979). Curriculum '78: Recommendations for the Undergraduate Program in Computer Science— a Report of the ACM Curriculum Committee on Computer Science [GS Search]. *Commun. ACM*, *22*(3), 147–166. https://doi.org/10.1145/359080.359083

Baniassad, E., Zamprogno, L., Hall, B., & Holmes, R. (2021). Stop the (autograder) insanity: Regression penalties to deter autograder overreliance [GS Search]. *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, 1062–1068. https://doi.org/10.1145/3408877.3432430

Barbosa, A., Costa, E., & Brito, P. (2023). Juízes online são suficientes ou precisamos de um var? [GS Search]. *Anais do III Simpósio Brasileiro de Educação em Computação*, 386–394. https://doi.org/10.5753/educomp.2023.228224

Becker, B. A., & Fitzpatrick, T. (2019). What do cs1 syllabi reveal about our expectations of introductory programming students? [GS Search]. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 1011–1017. https://doi.org/10.1145/3287324.3287485

Becker, B. A., Goslin, K., & Glanville, G. (2018). The effects of enhanced compiler error messages on a syntax error debugging test [GS Search]. *SIGCSE 2018 - Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, *2018-Janua*, 640–645. https://doi.org/10.1145/3159450.3159461

Bonwell, C., & Eison, J. (1991). *Active Learning: Creating Excitement in the Classroom* [GS Search]. Wiley.

Bosse, Y., & Gerosa, M. (2015). Reprovações e Trancamentos nas Disciplinas de Introdução à Programação da Universidade de São Paulo: Um Estudo Preliminar [GS Search]. *Anais do XXIII Workshop sobre Educação em Computação*, 426–435. https://doi.org/10.5753/wei.2015.10259

Brodley, C. E., Hescott, B. J., Biron, J., Ressing, A., Peiken, M., Maravetz, S., & Mislove, A. (2022). Broadening Participation in Computing via Ubiquitous Combined Majors (CS+X)

[GS Search]. *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, 544–550. https://doi.org/10.1145/3478431.3499352

Caceffo, R., Frank-Bolton, P., Souza, R., & Azevedo, R. (2019). Identifying and validating java misconceptions toward a cs1 concept inventory [GS Search]. *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, 23–29. https://doi.org/10.1145/3304221.3319771

Caceffo, R., Wolfman, S., Booth, K. S., & Azevedo, R. (2016). Developing a computer science concept inventory for introductory programming [GS Search]. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 364–369. https://doi.org/10.1145/2839509.2844559

Cain, A., & Babar, M. A. (2016). Reflections on applying constructive alignment with formative feedback for teaching introductory programming and software architecture [GS Search]. *Proceedings of the 38th International Conference on Software Engineering Companion*, 336–345. https://doi.org/10.1145/2889160.2889185

Cohen, L., Manion, L., & Morrison, K. (2005). *Research Methods in Education* [GS Search]. Routledge.

Crouch, C. H., & Mazur, E. (2001). Peer Instruction: Ten years of experience and results [GS Search]. *American Journal of Physics*, *69*(9), 970–977. https://doi.org/10.1119/1.1374249

De Ruvo, G., Tempero, E., Luxton-Reilly, A., Rowe, G. B., & Giacaman, N. (2018). Understanding semantic style by analysing student code [GS Search]. *Proceedings of the 20th Australasian Computing Education Conference*, 73–82. https://doi.org/10.1145/3160489.3160500

Dodds, Z., Libeskind-Hadas, R., & Bush, E. (2010). When CS 1 is Biology 1: Crossdisciplinary Collaboration as CS Context [GS Search]. *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, 219–223. https://doi.org/10.1145/1822090.1822152

Edwards, S. H. (2021). Automated feedback, the next generation: Designing learning experiences [GS Search]. *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, 610–611. https://doi.org/10.1145/3408877.3437225

Ellis, M., Shaffer, C. A., & Edwards, S. H. (2019). Approaches for coordinating etextbooks, online programming practice, automated grading, and more into one course [GS Search]. *SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 126–132. https://doi.org/10.1145/3287324.3287487

Fisler, K., Krishnamurthi, S., & Siegmund, J. (2016). Modernizing plan-composition studies [GS Search]. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 211–216. https://doi.org/10.1145/2839509.2844556

Galvão, L., Fernandes, D., & Gadelha, B. (2016). Juiz online como ferramenta de apoio a uma metodologia de ensino híbrido em programação [GS Search]. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação - SBIE)*, *27*(1), 140. https://doi.org/10.5753/cbie.sbie.2016.140

Gama, G., Caceffo, R., Souza, R., Bennati, R., Aparecida, T., Garcia, I., & Azevedo, R. (2018, November). *An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in Python* (tech. rep. No. IC-18-19) (GS Search). Institute of Computing, University of Campinas.

Guo, P. (2014). Python Is Now the Most Popular Introductory Teaching Language at Top U.S. Universities [Retrieved on 06/26/2023 from link].

Hollingsworth, J. (1960). Automatic graders for programming classes [GS Search]. *Commun. ACM*, *3*(10), 528–529. https://doi.org/10.1145/367415.367422

Hsu, S., Li, T. W., Zhang, Z., Fowler, M., Zilles, C., & Karahalios, K. (2021). Attitudes surrounding an imperfect ai autograder [GS Search]. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/3411764.3445424

Ihantola, P., & Petersen, A. (2019). Code complexity in introductory programming courses [GS Search]. *Proceedings of the 52nd Hawaii International Conference on System Sciences*, 7662–7670.

Inside Higher Ed. (2018, November). Autograder issues upset students at berkeley [Retrieved on 06/22/2023 from link].

Joni, S.-N. A., & Soloway, E. (1986). But My Program Runs! Discourse Rules for Novice Programmers [GS Search]. *Journal of Educational Computing Research*, *2*(1), 95–125. https://doi.org/10.2190/6E5W-AR7C-NX76-HUT2

Keuning, H., Heeren, B., & Jeuring, J. (2019). How teachers would help students to improve their code [GS Search]. *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, 119–125. https://doi.org/10.1145/3304221.3319780

Keuning, H., Heeren, B., & Jeuring, J. (2021). A tutoring system to learn code refactoring [GS Search]. *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, 562–568. https://doi.org/10.1145/3408877.3432526

Kinnunen, P., & Malmi, L. (2006). Why students drop out cs1 course? [GS Search]. *Proceedings of the Second International Workshop on Computing Education Research*, 97–108. https://doi.org/10.1145/1151588.1151604

Kluvyer, T. (n.d.). Green Tree Snakes - the missing Python AST docs [Retrieved on 06/05/2023 from link].

Lazar, J., Feng, J. H., & Hochheiser, H. (2017). *Research Methods in Human-Computer Interaction* [GS Search]. Morgan Kaufmann.

Lima, M. A. P., Carvalho, L. S. G., Oliveira, E. H. T., Oliveira, D. B. F., & Pereira, F. D. (2021). Uso de atributos de código para classificar a dificuldade de questões de programação em juízes online [GS Search]. *Revista Brasileira de Informática na Educação*, *29*, 1137–1157. https://doi.org/10.5753/rbie.2021.29.0.1137

Liu, D., & Petersen, A. (2019). Static analyses in python programming courses [GS Search]. *SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 666–671. https://doi.org/10.1145/3287324.3287503

Marwan, S., Lytle, N., Williams, J. J., & Price, T. (2019). The impact of adding textual explanations to next-step hints in a novice programming environment [GS Search]. *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, 520–526. https://doi.org/10.1145/3304221.3319759

Oliveira, E., Keuning, H., & Jeuring, J. (2023). Student code refactoring misconceptions [GS Search]. *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, 19–25. https://doi.org/10.1145/3587102.3588840

Pereira, F. D., Oliveira, E. H. T., Oliveira, D. B. F., Cristea, A. I., Carvalho, L. S. G., Fonseca, S. C., Toda, A., & Isotani, S. (2020). Using learning analytics in the amazonas: Understanding students' behaviour in introductory programming [GS Search]. *British Journal of Educational Technology*, *51*(4), 955–972. https://doi.org/https://doi.org/10.1111/bjet.12953

Pereira, F. D., Fonseca, S. C., Oliveira, E. H., Oliveira, D. B., Cristea, A. I., & Carvalho, L. S. (2020). Deep learning for early performance prediction of introductory programming students: A comparative and explanatory study [GS Search]. *Revista Brasileira de Informática na Educação*, *28*(0), 723–748. https://doi.org/10.5753/rbie.2020.28.0.723

Porfirio, A. J., Pereira, R., & Maschio, E. (2021). A-Learn EvId: A Method for Identifying Evidence of Computer Programming Skills Through Automatic Source Code Assessment [GS Search]. *Revista Brasileira de Informática na Educação*, *29*, 692–717. https://doi.org/10.5753/rbie.2021.29.0.692

Porter, L., Bailey Lee, C., & Simon, B. (2013). Halving fail rates using peer instruction: A study of four computer science courses [GS Search]. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, 177–182. https://doi.org/10.1145/2445196.2445250

Prather, J., Pettit, R., McMurry, K., Peters, A., Homer, J., & Cohen, M. (2018). Metacognitive difficulties faced by novice programmers in automated assessment tools [GS Search]. *ICER 2018 - Proceedings of the 2018 ACM Conference on International Computing Education Research*, 41–50. https://doi.org/10.1145/3230977.3230981

Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review [GS Search]. *ACM Transactions on Computing Education (TOCE).*, *18*(1). https://doi.org/10.1145/3077618

Silva, E., Caceffo, R., & Azevedo, R. (2021). Análise estática de código em conjunto com autograders [GS Search]. *Anais Estendidos do I Simpósio Brasileiro de Educação em Computação*, 25–26. https://doi.org/10.5753/educomp_estendido.2021.14858

Silva, E., Caceffo, R., & Azevedo, R. (2023a). *Misconceptions in Correct Code: rating the severity of undesirable programming behaviors in Python CS1 courses* (tech. rep. No. IC-23-01) (GS Search). Institute of Computing, University of Campinas. https://doi.org/10.13140/RG.2.2.28739.89127

Silva, E., Caceffo, R., & Azevedo, R. (2023b). Passar nos casos de teste é suficiente? identificação e análise de problemas de compreensão em códigos corretos [GS Search]. *Anais do III Simpósio Brasileiro de Educação em Computação*, 119–129. https://doi.org/10.5753/educomp.2023.228346

Silva, E., Caceffo, R., & Azevedo, R. (2023c). A syllabi analysis of cs1 courses from brazilian public universities [GS Search]. *Brazilian Journal of Computers in Education*, *31*(1), 407–436. https://doi.org/10.5753/rbie.2023.2870

Simon, B., Esper, S., Porter, L., & Cutts, Q. (2013). Student experience in a student-centered peer instruction classroom [GS Search]. *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, 129–136. https://doi.org/10.1145/2493394.2493407

Simon, B., Kohanfars, M., Lee, J., Tamayo, K., & Cutts, Q. (2010). Experience report: Peer instruction in introductory computing [GS Search]. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, 341–345. https://doi.org/10.1145/1734263.1734381

Simon, B., Parris, J., & Spacco, J. (2013). How we teach impacts student learning: Peer instruction vs. lecture in cs0 [GS Search]. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, 41–46. https://doi.org/10.1145/2445196.2445215

Sloan, R. H., Taylor, C., & Warner, R. (2017). Initial Experiences with a CS + Law Introduction to Computer Science (CS 1) [GS Search]. *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, 40–45. https://doi.org/10.1145/3059009.3059029

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge [GS Search]. *IEEE Transactions on Software Engineering*, *SE-10*(5), 595–609. https://doi.org/10.1109/TSE.1984.5010283

Sorva, J., & Vihavainen, A. (2016). Break statement considered [GS Search]. *ACM Inroads*, *7*(3), 36–41. https://doi.org/10.1145/2950065

Tew, A. E., & Guzdial, M. (2011). The fcs1: A language independent assessment of cs1 knowledge [GS Search]. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, 111–116. https://doi.org/10.1145/1953163.1953200

Ureel II, L. C., & Wallace, C. (2019). Automated critique of early programming antipatterns [GS Search]. *SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 738–744. https://doi.org/10.1145/3287324.3287463

VanLehn, K. (2006). The behavior of tutoring systems [GS Search]. *International journal of artificial intelligence in education*, *16*(3), 227–265.

Walker, H. M. (2017). ACM RETENTION COMMITTEE Retention of Students in Introductory Computing Courses: Curricular Issues and Approaches [GS Search]. *ACM Inroads*, *8*(4), 14–16. https://doi.org/10.1145/3151936

Wichmann, B. A., Canning, A., Clutterbuck, D., Winsborrow, L., Ward, N., & Marsh, D. W. R. (1995). Industrial perspective on static analysis [GS Search]. *Software Engineering Journal*, *10*, 69–75(6).