Universidade Estadual de Campinas
Instituto de Computação

Eryck Pedro da Silva

Misconceptions in Correct Code: Assisting Instructors and Students by Shedding Light on What is Potentially Overshadowed by Automated Correction

Problemas de Compreensão em Códigos Corretos: Auxiliando Instrutores e Alunos ao Iluminar o que é Potencialmente Ofuscado pela Correção Automática

CAMPINAS
2024

Eryck Pedro da Silva

Misconceptions in Correct Code: Assisting Instructors and Students by Shedding Light on What is Potentially Overshadowed by Automated Correction

Problemas de Compreensão em Códigos Corretos: Auxiliando Instrutores e Alunos ao Iluminar o que é Potencialmente Ofuscado pela Correção Automática

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Rodolfo Jardim de Azevedo
Co-supervisor/Coorientador: Dr. Ricardo Edgard Caceffo

Este exemplar corresponde à versão final da Tese defendida por Eryck Pedro da Silva e orientada pelo Prof. Dr. Rodolfo Jardim de Azevedo.

CAMPINAS
2024

Ficha catalográfica
Universidade Estadual de Campinas (UNICAMP)
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Si38m

Silva, Eryck Pedro da, 1993-
Misconceptions in correct code : assisting instructors and students by shedding light on what is potentially overshadowed by automated correction / Eryck Pedro da Silva. – Campinas, SP : [s.n.], 2024.

Orientador: Rodolfo Jardim de Azevedo.
Coorientador: Ricardo Edgard Caceffo.
Tese (doutorado) – Universidade Estadual de Campinas (UNICAMP), Instituto de Computação.

1. Programação (Computadores) - Estudo e ensino. 2. Aprendizagem - Avaliação. 3. Python (Linguagem de programação de computador). I. Azevedo, Rodolfo Jardim de, 1974-. II. Caceffo, Ricardo Edgard, 1983-. III. Universidade Estadual de Campinas (UNICAMP). Instituto de Computação. IV. Título.

Informações Complementares

Título em outro idioma: Problemas de compreensão em códigos corretos : auxiliando instrutores e alunos ao iluminar o que é potencialmente ofuscado pela correção automática
Palavras-chave em inglês:
Computer programming - Study and teaching
Assessment of learning
Python (Computer program language)
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Rodolfo Jardim de Azevedo [Orientador]
Rodrigo Silva Duran
Leandro Silva Galvão de Carvalho
Jacques Wainer
Julio Cesar dos Reis
Data de defesa: 01-08-2024
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-5141-6936
- Currículo Lattes do autor: http://lattes.cnpq.br/4106945354159329

Universidade Estadual de Campinas
Instituto de Computação

Eryck Pedro da Silva

Misconceptions in Correct Code: Assisting Instructors and Students by Shedding Light on What is Potentially Overshadowed by Automated Correction

Problemas de Compreensão em Códigos Corretos: Auxiliando Instrutores e Alunos ao Iluminar o que é Potencialmente Ofuscado pela Correção Automática

Banca Examinadora:

- Prof. Dr. Rodolfo Jardim de Azevedo
  IC/UNICAMP
- Prof. Dr. Rodrigo Silva Duran
  IFMS
- Prof. Dr. Leandro Silva Galvao de Carvalho
  IComp/UFAM
- Prof. Dr. Jacques Wainer
  IC/UNICAMP
- Prof. Dr. Julio Cesar dos Reis
  IC/UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 01 de agosto de 2024

Dedication

I dedicate this thesis to my father, Edivaldo. One of the first things I remember you teaching me was how to count money. You told me to start with the larger values and then add the smaller ones. Little did I wonder back then that you were teaching me a metaphor on how life should be lived: solving bigger problems first makes the smaller ones simpler. This teaching, amongst many others, helped me endure the rainy periods throughout this journey. Today, paraphrasing Elidibus[1]*, I am proud to say that the rains have ceased, and I am grateful for the view of a beautiful day. Though you are not here to see it, I am sure you are witnessing it from wherever you are, and one day, at duty's end, we will meet again.*

[1] Character in FINAL FANTASY XIV Online, a game developed by Square Enix.

You are stronger than all things that made you weak.

("You Are Stronger". Music written by Lotus Juice.)

Acknowledgements

To God, for allowing me to conduct this work with the support of my family and friends.

To my parents, Edivaldo (in memoriam) and Cícera, for believing in me even during the times when I doubted myself. Your love and guidance will always be among the main reasons for my achievements.

To my sister, Monica, and my niece, Isabelly, for always providing me with support and motivation. I know that no matter how challenging life gets, we will always be there for each other.

To my advisor, Rodolfo, and co-advisor, Ricardo, for all their wisdom and, more importantly, their patience in assisting me throughout these years. You can be certain that the things I learned were not only for this Ph.D. but for life as well.

To professors Tomasz, Zanoni, Paulo, and Jacques, as well as the MC102 teaching assistants, for their immense support in conducting my research at UNICAMP. I also thank all 32 CS1 instructors and 56 undergraduates who volunteered for the studies.

To all my friends at LSC for welcoming me so openly when I first entered that laboratory. The laughter, escapades, and, of course, coffee, certainly aided me on this journey.

To Victor and Paulo, my long-term friends since middle school. We have stuck together through challenges, setbacks, and victories. Your companionship was, and will always be, an essential part of my life.

To Lucas Oliveira, for immensely assisting me in developing the didactic materials used in this thesis. Your support was crucial during times when I was almost having a mental breakdown trying to understand how to edit the short videos.

To Leonardo Lima, my dearest friend, words will never be enough to describe what you have done for me. God arranged for us to meet during the most challenging time of my life, and we persevered. Thank you for all the text revisions. Thank you for all the Google Meet conversations which always lasted at least five hours. Thank you for enduring my whims and tantrums despite being in a completely different academic field. Thank you, for everything.

To Bp. Bruno Leonardo, for sharing your wisdom in God's name. Your words strengthened my faith during the periods when I needed them the most. My bishop, have I told you today that I love you?

To all brothers, sisters, and entities at F ∴ E ∴ E ∴ U whom helped me immensely during the times when not only my body ached, but my spirit as well. Both healing and wisdom were key factors that helped me continue on this journey.

And to all my other friends and family members who supported me during this long and challenging research period.

This work was partially funded by the Teaching, Research and Extension Support Fund (FAEPEX) under grant numbers 38813-20 and 69086-24, and by the Brazilian National Council for Scientific and Technological Development (CNPq) under grant number 142476/2020-0.

Resumo

Um dos desafios do ensino-aprendizagem de introdução à programação em universidades (CS1) é a elaboração de *feedback* adequado aos estudantes: um elevado número de alunos por turma gera dificuldades ou mesmo inviabiliza a interação individual entre instrutor e discente. Nesse cenário, a geração de *feedback* costuma ser realizada por meio de sistemas de avaliação automática de código. No entanto, como essas ferramentas costumam apenas verificar se o código está correto, outras características presentes podem ser ignoradas. Nesta tese, foram identificados Problemas de Compreensão em Códigos Corretos (PC^3), que são comportamentos que potencialmente indicam compreensões incompletas ou erradas sobre os conceitos de CS1. Ao todo, 45 PC^3 foram identificados ao analisar manualmente 2,441 códigos de alunos em uma turma de introdução à programação em Python lecionada no paradigma estruturado. A validação inicial dos PC^3 foi realizada por meio de consulta docente e discente, com o objetivo de identificar as potenciais causas dos alunos cometerem esses comportamentos. A consulta com os docentes também permitiu listar 15 PC^3 considerados como mais graves, ou seja, que possuem maior prioridade de correção em sala de aula. Com foco da tese redirecionado para esses PC^3 mais graves, foram construídos artefatos educacionais para auxiliar o ensino e a aprendizagem de CS1 com respeito a esses comportamentos: uma ferramenta de detecção automática e a elaboração de materiais didáticos que abordam a explicação do porquê esses PC^3 devem ser evitados. A ferramenta de detecção automática foi utilizada em um estudo em larga escala para avaliar a frequência dos PC^3 mais graves ao longo de oito semestres letivos de uma disciplina de CS1, totalizando mais de 40.000 submissões. Os resultados obtidos demonstram que os PC^3 mais graves ocorrem continuamente ao longo de um semestre letivo, não aparentando serem corrigidos por conta própria dos alunos. Oito desses PC^3 mais graves englobam tópicos sobre comandos de decisão e de repetição, assuntos cruciais em CS1. O restante denota uma despreocupação discente em manter organização e legibilidade dos códigos. Os materiais educacionais foram avaliados em um estudo de caso com 23 discentes de uma turma de CS1. Os resultados obtidos sugerem que a instrução, dentro ou fora de sala de aula, tem o poder de influenciar tanto a ocorrência como a correção de PC^3. No entanto, somente a aplicação de materiais não aparenta ser suficiente para mitigar PC^3, pois o ambiente educacional precisa reforçar, em mais aspectos, a necessidade dos discentes evitarem esses comportamentos. Esta tese propõe que docentes e discentes de disciplinas de introdução à programação atentem-se para a verificação dos PC^3 além da corretude dos códigos, pois os resultados sugerem que os alunos obtêm aprovação nessas disciplinas, podendo atingir a nota máxima, mas ainda assim carregar uma compreensão incompleta ou errada dos conceitos aprendidos.

Abstract

One of the challenges with respect to teaching and learning undergraduate introductory programming courses (CS1) is developing appropriate feedback for students. This challenge is especially seen in classes with a high number of students, as this situation significantly hinders the individual interaction between the instructor and the student. In this context, feedback generation is usually carried out through automatic code evaluation systems. However, since these tools generally check only whether the code is correct, other features present in the code may be ignored. This thesis identified Misconceptions in Correct Code (MC^3), coding behaviors present in novice programmers' code that potentially indicate incomplete or incorrect understanding of CS1 concepts. In total, 45 MC^3 were identified by manually analyzing 2,441 student codes from a Python CS1 course taught within the structured programming paradigm. The initial validation of MC^3 was conducted by consulting both CS1 teachers and students, to identify potential causes of why students develop these coding behaviors. Consulting with CS1 teachers also allowed listing 15 most severe MC^3, i.e., those with a greater correction priority in the classroom. Shifting the thesis focus to these most severe MC^3, educational artifacts were constructed to assist in CS1 teaching and learning with respect to these behaviors: a tool for automated detection and elaboration of didactic materials that explains about these MC^3 and informs why they should be avoided. The automated detection tool was used in a large-scale study to assess the frequency of the most severe MC^3 in eight academic terms of a CS1 course, totaling more than 40,000 submissions. The results obtained demonstrate that MC^3 occurs continuously throughout an academic semester, causing students to not correct these behaviors by themselves. Eight of these severe MC^3 encompasses topics related to decision and iteration structures, crucial concepts in CS1. The remaining evidence a careless approach to coding, as students tend to ignore aspects such as organization and readability. The didactic materials were evaluated in a case study with 23 students in a CS1 class. The evidence collected suggests that instruction, inside or outside the classroom, has the power to influence both the occurrence and mitigation of MC^3. However, using the materials appears not to be enough to effectively mitigate the occurrence of MC^3. The educational environment must reinforce, in more ways, the reasons why students should avoid these coding behaviors. This thesis proposes that both CS1 teachers and students should also focus on addressing MC^3 in addition to code correctness, as the results suggest that students often obtain approval in these courses, possibly even with a maximum grade, but still carry an incomplete or wrong understanding of the concepts learned.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

"All the journeys start some where with a first step."
—"**Full Moon Full Life**". Music written by Lotus Juice.

I begin this thesis by stating that its genesis traces back to my undergraduate years, where the seeds of my academic pursuit began to sprout. Though the early stages of my journey were unassuming, the challenges inherent in the pursuit of a bachelor's degree in computer science played a slow but pivotal role in igniting a somewhat latent desire to become a researcher. However, it was not until the culmination of my undergraduate studies that I recognized a profound interest in the intersection of computing and education, particularly in how computational tools could enhance the teaching and learning experience, especially for students outside the field of Computer Science (CS). This realization coalesced my personal motivations, propelling me towards a deeper exploration of this research domain.

The integration of CS programs into university curricula has often been justified by the need to cultivate a new generation of highly skilled professionals, particularly in technical domains like programming [60]. Traditionally, these programs have drawn students from Science, Technology, Engineering, and Mathematics (STEM) fields. However, there exists another compelling argument grounded in the demands of the labor market, which emphasizes the necessity for professionals that are adept not only in programming but also in leveraging CS knowledge to automate tasks such as spreadsheet management, database access, and interactive art creation [21]. This argument portrays an interdisciplinary appeal of CS, leading to an extension beyond STEM disciplines, permeating into diverse areas such as Law [123], Cognitive Psychology [25], and Biology [40]. Consequently, CS educators often find themselves catering to student cohorts from non-STEM backgrounds[1]. This situation, in my opinion, elicits a distinct set of interesting challenges that needs to be addressed both inside and outside the classroom.

This thesis centers on a specific aspect of Computer Science education: introductory programming. Within the literature, undergraduate courses covering this subject are

[1]I acknowledge literature often refers to these groups of students as *majors* and *non-majors* in CS. However, I am not aware of any related term used in Brazilian's higher education system. Therefore, I chose not to use the literature terms as they might induce similarity between different types of higher education systems.

commonly referred to as Computer Science 1 (CS1) courses [8, 62]. Despite the absence of consensus regarding the precise content of CS1 courses [13, 14, 62, 115], they are designed to cultivate students' systematic and logical reasoning skills. Moreover, these courses often serve as the initial exposure to programming languages, particularly for students enrolled in non-STEM programs. Consequently, CS1 courses play a foundational role in undergraduate curricula, shaping students' preparedness for subsequent courses and their future professional endeavors [33, 91, 125]. I personally believe this applies to all students who take the CS1 course, independently of their undergraduate program. Once again, the challenge remains to CS educators to foster this vision upon their STEM and non-STEM students.

The significance of CS1 courses is accompanied by a multitude of challenges inherent in their teaching and learning processes. These challenges manifest in high rates of failure and dropout, as documented in both global [76, 90, 98, 140] and Brazilian national contexts [24, 32, 53]. Contributing factors to these rates include the large class sizes, which hinder individualized attention between students and instructors, and the misalignment of student expectations with the course content (e.g., the misconception that CS can be entirely detached from mathematics) [20]. Additionally, the diverse educational backgrounds of students prior to enrolling in the undergraduate program also elicits these rates [140]. This factor can be related to students' performance in STEM courses in primary and secondary education [32] as well as varying levels of computer literacy, since many students may only have developed skills such as text editing or web browsing.

Reasons for student dropout in CS1 courses often stem from a combination of time constraints and a lack of motivation to continue the course [76]. This lack of motivation derives from students' view of CS1 as a challenging subject, thus leading to difficulties in efficiently allocating study time to grasp course concepts. Additionally, Galvão et al. [53] stated that CS1 courses are frequently viewed as peripheral activities within non-STEM undergraduate programs, fostering the misconception that programming skills are merely obligatory components of the curriculum rather than essential tools for future professional endeavors. Moreover, even students that do not evade from CS1 may experience feelings of inadequacy compared to their peers, indicating a pervasive sense of falling behind [98].

The utilization of automatic grading systems (autograders) represents another aspect addressed in this thesis. In the context of CS1 courses, autograders have been employed since 1960 to accommodate larger class sizes, offering significant time and cost savings [65]. In contemporary settings, Massive Open Online Courses (MOOCs) frequently utilize autograders to bridge the gap in formative feedback often lacking in online learning environments [88]. However, the use of autograders can also lead to unintended consequences. These consequences often manifest in behaviors that negatively impact the overall learning outcomes of CS1 courses. For instance, some students may exhibit a tendency to excessively rely on the feedback generated by autograders [10, 66]. Another problematic behavior emerges when students prioritize satisfying the autograder's criteria for correctness at the expense of neglecting other essential aspects of coding. This thesis specifically focuses on addressing the latter behavior.

1.1 Problem Definition

Assessment is the primary means by which instructors determine whether students are meeting the learning outcomes of a given subject [79]. These assessments can be designed as formative, to gauge learning progress, or summative, to evaluate the achievement of learning objectives. In computer education research, much attention has been devoted to the assessment of programming exercises. However, as Lancaster et al. [79] note, despite the recognition of assessment's vital role in achieving learning outcomes, there is no consensus on how it should be structured. This lack of agreement stems from the complex and multifaceted nature of programming, which encompasses both the cognitive load placed on students and the varying levels of intuition instructors have about programming. Additionally, the ongoing debate over how programming courses should be taught contributes to the lack of consensus [62, 107].

Given the importance of assessment in education, the question arises of how it can be effectively implemented in the classroom. One solution has been the design of conducting assessment in an automated format. In CS1, this approach restricts assessment to objective tasks, such as multiple-choice questionnaires, program correctness based on a predetermined set of expected output, and adherence to specific coding styles [85, 86, 103]. While the use of autograders saves time and allows for larger class sizes [65], a significant drawback is the limited feedback provided, especially when compared to those from a human instructor [45, 79].

Prather et al. [103] categorize autograders into three main classifications based on their functionalities: assessment of code output, promotion of test-based development, and support for students enrolled in programming courses. These tools extend beyond classroom environments; for instance, Competitive Programming [78] platforms attract both novice and experienced programmers to hone their problem-solving skills. In such contexts, autograders evaluate code based on factors beyond correctness, including running time. Platforms like Beecrowd[2] and Codeforces[3] employ grading systems that rank competitors, fostering a competitive environment reminiscent of gaming. Participants progress to more challenging problem sets upon completion of earlier ones, enhancing engagement and skill development.

As previously mentioned, autograders that solely assess code output play a crucial role in assisting instructors and teaching assistants by reducing the workload associated with assessment [46, 53, 65]. However, like any tool, autograders are prone to failure. Since students typically need to submit their solutions via the autograder system, frustration can quickly escalate if the system malfunctions, particularly when this happens close to an assignment deadline [68]. I can personally attest to this since I received my fair share of e-mails from my students in a late Sunday's night alleging the submission system was offline and there was nothing they could do. Another issue arises from an excessive reliance on autograder feedback [10]. In such cases, students tend to rely solely on error messages from the autograder when elaborating their solutions, foregoing a more effort-

[2] `https://judge.beecrowd.com/`
[3] `https://codeforces.com/`

based approach to coding. This dependency is detrimental as it discourages students from actively engaging in the problem-solving process.

Autograders have played a significant role in research aimed at identifying and addressing code that fails to compile or achieve the desired outcome [7, 16, 83, 94]. Such research is important, considering that this type of code often represents students' initial programming attempts. However, in the context of CS1 classes, even code that produces the expected output may exhibit characteristics that experienced programmers would typically avoid [39, 75, 126]. This suggests that students who receive maximum grades from autograders may still possess incorrect or incomplete understandings of the concepts taught. In educational settings where neither instructors nor autograders assess code characteristics beyond correctness, students can successfully complete CS1 courses while harboring faulty comprehensions.

Examples of assessments that go beyond code correctness include evaluations of code quality [23, 86]. In these cases, students' code are assessed based on adherence to predetermined coding styles, such as indentation, variable naming, and function length [86]. Feedback on this type of assessment is valuable to students, as readability and maintainability are essential skills expected of professional programmers [23, 70, 73].

However, while such assessments focus on surface-level code quality, CS1 students may still write code with characteristics that suggest a faulty or incomplete understanding of key concepts, such as decision and iteration structures. These issues may not be detected by standard coding style checks. An instance of this type of incomplete comprehension regarding a CS1 topic is exemplified in Code 1.1. In a typical assignment task involving the classification of a triangle based on the lengths of its sides, a student initially checks if all three sides (`sA`, `sB`, and `sC`) are equal (line 5). Subsequently, the student uses an `elif` statement (line 8) to determine if the triangle is isosceles. However, in the same `elif` statement, the student also redundantly verifies if the previous condition (line 5) is false. Since the `elif` statement inherently performs this check, the second declared condition of the `elif` in line 8 is unnecessary. This redundancy suggests a potential misunderstanding of how the `elif` statement operates. Code 1.2 presents an amended version of Code 1.1, omitting the redundant check within the `elif` statement in line 8.

```
1  sA = float(input("Side␣A:"))
2  sB = float(input("Side␣B:"))
3  sC = float(input("Side␣C:"))
4
5  if sA == sB == sC:
6      print("Equilateral␣triangle.")
7
8  elif (sA == sB or sA == sC or sB == sC) and not (sA == sB == sC):
9      print("Isosceles␣triangle.")
10
11 else:
12     print("Scalene␣triangle.")
```

Code 1.1: Example of an `elif` retesting an already checked condition.

```
1  sA = float(input("Side␣A:"))
2  sB = float(input("Side␣B:"))
3  sC = float(input("Side␣C:"))
4
5  if sA == sB == sC:
6      print("Equilateral␣triangle.")
7
8  elif sA == sB or sA == sC or sB == sC:
9      print("Isosceles␣triangle.")
10
11 else:
12     print("Scalene␣triangle.")
```

Code 1.2: Code 1.1 without the unnecessary check in the `elif`.

Both Code 1.1 and 1.2 produce the expected outcome in an assessment solely based on output, earning students maximum grades. However, it is conceivable that a student may harbor incomplete or incorrect understandings of the `elif` statement in Code 1.1. This thesis directs its investigation towards coding behaviors that may indicate flawed comprehension, despite code correctness. Hence, these behaviors are termed Misconceptions in Correct Code (MC^3)[4]. In this context, I have chosen to classify them as misconceptions [106] since, within the CS1 domain, literature often defines misconceptions as syntactic, semantic, or logical errors [6, 28, 30, 54]. However, these studies do not limit their investigation to correct code.

The study on MC^3 presented in this thesis contributes to CS1 programming assessment research by extending the common focus on automated feedback generation, identified as the most widespread method [85]. The aim of this research was to uncover the underlying causes of MC^3 and develop educational materials designed to support instructors while providing students with diverse forms of feedback.

1.2 Objectives

The research on MC^3 outlined in this thesis aims to enhance the teaching and learning process for both students and instructors of CS1 courses. This study will specifically target CS1 courses utilizing the imperative programming paradigm, due to constraints of scope. The target audience comprises students, whom the research aims to empower by providing tools to identify and understand why MC^3 should be avoided; and instructors, who will benefit from the identification and categorization of MC^3, along with insights into the reasons behind student development of these behaviors. In light of these factors, the hypothesis and both main and specific objectives of this thesis were established.

1.2.1 Hypothesis

This research hypothesizes that CS1 students can write code that produces correct outputs while still harboring misconceptions about the underlying programming concepts.

[4] In Brazilian Portuguese these were named as *Problemas de Compreensão em Códigos Corretos*, thus leading to *PC^3* as its acronym.

Furthermore, these misconceptions can be systematically identified, cataloged, and effectively addressed in CS1 classes.

1.2.2 Main Objective

The main objective of this thesis is to identify, assess, and catalog MC³ within CS1 environments. Through this process, the groundwork lays the development of educational artifacts aimed at addressing these misconceptions. These artifacts serve within and beyond CS1 classrooms, aiding both instructors and students in raising awareness about MC³. By gaining an understanding of MC³, students are likely to deepen their comprehension of CS1 concepts, thereby mitigating the occurrence of these behaviors in their code.

1.2.3 Specific Objectives

SO1: To systematically identify, analyze, and validate MC³ within CS1 environments. The validation process encompasses two contexts: internal and external. The internal context pertains to the specific environment where the research on MC³ is conducted, ensuring the relevance and applicability of the findings within this setting. Moreover, the external context examines how the identified MC³ may manifest in CS1 classes across similar teaching contexts, thereby establishing the generalizability of the thesis' findings.

SO2: To develop valid artifacts designed to address MC³ within CS1 classes. These artifacts serve to aid both instructors and students in identifying and comprehending the underlying causes of MC³, as well as providing guidance on how to avoid them when coding.

SO3: To implement and assess the artifacts developed in SO2 within a CS1 class. The assessment involves gathering reflections on the adoption of these artifacts, including students' perceptions and engagement. Additionally, the assessment aims to determine whether the artifacts effectively mitigate the occurrence of MC³ among students.

1.3 Contributions

This thesis makes the following contributions:

- It conducts a comprehensive review of CS1 syllabi in Brazilian Public Universities, elucidating the most common teaching contexts. (Chapter 2)
- It identifies and validates 45 MC³ through manual assessment of students' code. Subsequently, 15 MC³ are further analyzed since these are classified most needing addressment in CS1 classes. (Chapter 3)
- It presents an automated detection tool for 14 of the 15 MC³ classified as most in need of addressment in CS1 classes. The tool is developed in Python using static code analysis. (Chapter 3)

- It presents educational materials comprised of lecture slides, short videos, and flashcards for nine of the 14 automatically detectable MC^3 since these nine were identified as most common (Chapter 4).

This thesis claims that even though MC^3 exhibit varying frequency distributions, they do not entirely vanish by the conclusion of a CS1 course. Consequently, students may complete the course while retaining misconceptions about learned concepts. Furthermore, the study finds that MC^3 occurrence is not dependent on previous occurrences, nor are influenced by time constraints regarding the submission deadlines for assignments. However, instructional methods, both within and outside CS1 classes, appear capable of both inducing and mitigating the incorporation of MC^3. The assessment of educational artifacts underscores the importance of elaborating MC^3 educational materials that engage students by addressing language barriers effectively and presenting content in a format that captures students' attention. As correctness remains the primary goal for students in assignment solutions, materials that fail to motivate them to push themselves beyond correctness may risk being perceived as unimportant.

The elaboration of this thesis resulted in the following publications in chronological order:

1. Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. Análise estática de código em conjunto com autograders. In *Anais Estendidos do I Simpósio Brasileiro de Educação em Computação*, pages 25–26, Porto Alegre, RS, Brasil, 2021. SBC. DOI: `https://doi.org/10.5753/educomp_estendido.2021.14858` [114].

2. Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. Análise dos Tópicos Mais Abordados em Disciplinas de Introdução à Programação em Universidades Federais Brasileiras. In *Anais do II Simpósio Brasileiro de Educação em Computação*, pages 29–39, Porto Alegre, RS, Brasil, 2022. SBC. DOI: `https://doi.org/10.5753/educomp.2022.19196` [115].

3. Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. Misconceptions in Correct Code: rating the severity of undesirable programming behaviors in Python CS1 courses. Technical Report IC-23-01, Institute of Computing, University of Campinas, 2023. DOI: `http://dx.doi.org/10.13140/RG.2.2.28739.89127` [116].

4. Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. Passar nos casos de teste é suficiente? Identificação e análise de Problemas de Compreensão em Códigos Corretos. In *Anais do III Simpósio Brasileiro de Educação em Computação*, pages 119–129, Porto Alegre, RS, Brasil, 2023. SBC. DOI: `https://doi.org/10.5753/educomp.2023.228346` [117].

5. Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. A syllabi analysis of CS1 courses from brazilian public universities. *Brazilian Journal of Computers in Education*, 31 (1):407–436, Aug. 2023. DOI: `https://doi.org/10.5753/rbie.2023.2870` [118].

6. Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. When Test Cases Are Not Enough: Identification, Assessment, and Rationale of Misconceptions in Correct Code (MC^3). *Brazilian Journal of Computers in Education*, 31:1165–1199, Dec. 2023. DOI: https://doi.org/10.5753/rbie.2023.3552 [119].

Publication 2 was awarded as Best Track Paper in the symposium, resulting in Publication 5 as its extended version. Similarly, Publication 4 received an Honorable Mention in the symposium, resulting in Publication 6 as its extended version.

1.4 Ethical Considerations

Since part of the conducted research of this thesis involved human participants, all projects received prior evaluation and approval from the Ethics Research Committee affiliated with Universidade Estadual de Campinas. A total of three projects were elaborated. Approval of all three projects is present in Appendix D.

- **Title**: *Análise de Problemas de Programação em Disciplinas de Introdução à Programação em Python.*

  Title (English): *An Analysis of Programming Issues in Python Introductory Programming Courses.*

  CAAE: 51444121.5.0000.5404.

- **Title**: *Aplicação de uma Observação Semiestruturada em Turma de Introdução à Programação.*

  Title (English): *Application of a Semistructured Observation in an Introductory Programming Course.*

  CAAE: 60258622.8.0000.5404.

- **Title**: *Aplicação e Avaliação de Artefatos Educacionais em Turma de Introdução à Programação.*

  Title (English): *Application and Assessment of Educational Artifacts in an Introductory Programming Course.*

  CAAE: 70220523.0.0000.5404.

1.5 Methodology and Text Organization

This thesis adopts the format of an article collection, a structure endorsed by Universidade Estadual de Campinas[5]. In this format, the thesis is organized into chapters consisting of published or to-be-published articles. For the purpose of this thesis, I have opted to present the extended versions (publications 5 and 6) of the published articles, as they offer a more comprehensive exposition of the research findings.

[5] https://www.prpg.unicamp.br/wp-content/uploads/sites/10/2021/06/ccpg_in_2021_002_20210616.pdf

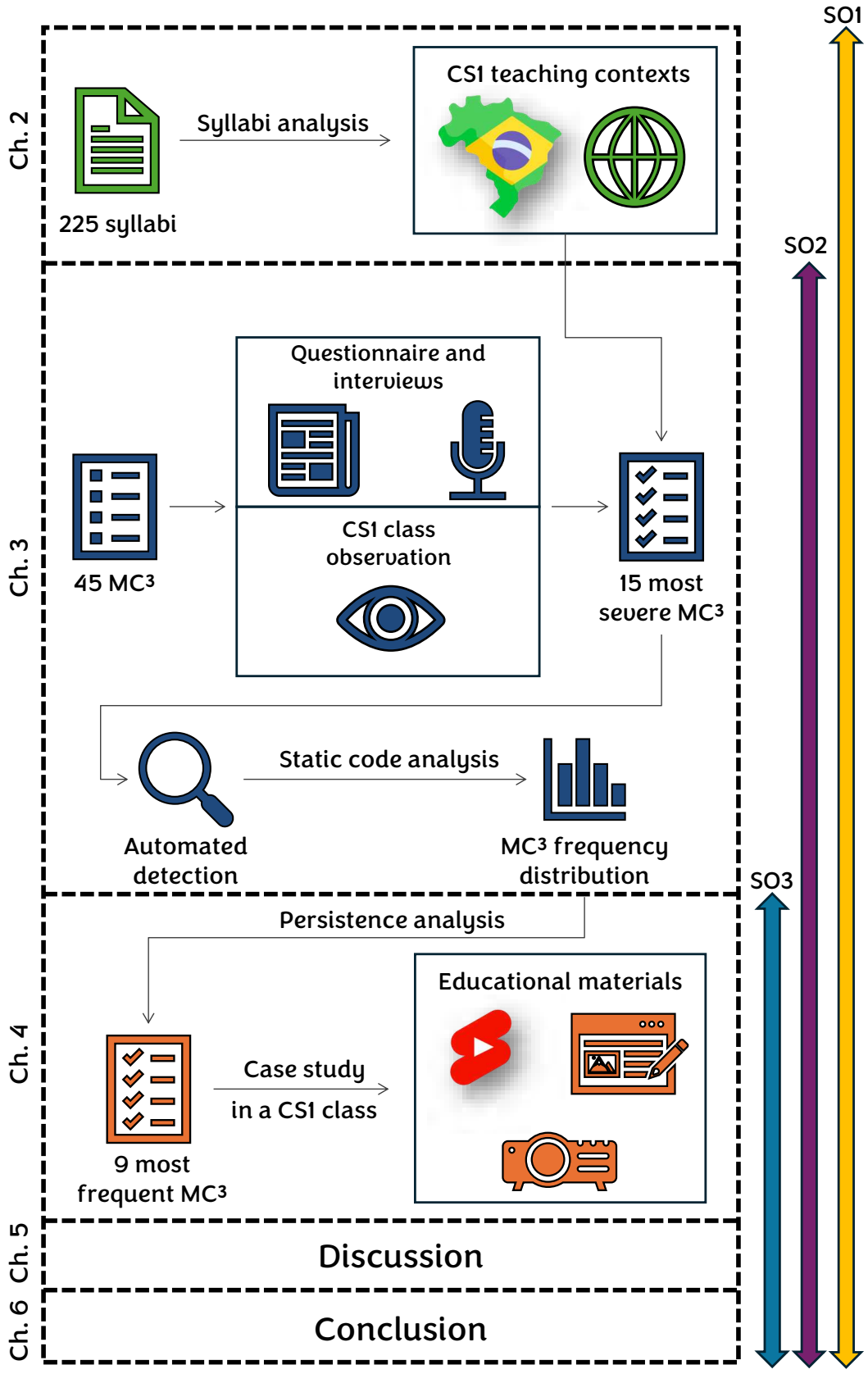

Figure 1.1: Summary of the methodology used and the obtained results of this thesis.

The research questions derived from the objectives of this thesis are presented in Chapters 2, 3, and 4. Additionally, while the text of these chapters were elaborated as they were published, I have added afterword sections to them with the purpose of clarity and connection to this thesis. Publisher authorization for Chapters 2 and 3 can be found in Appendix C. Figure 1.1 presents a summary of the methodology used alongside the results obtained in each chapter. The connection between the results of each lane are demonstrated as well as the corresponding Specific Objective (SO1, SO2, and SO3) assessed in each chapter.

Chapter 2 presents an analysis of CS1 syllabi in Brazil, aiming to elucidate the landscape of the teaching of these courses within the country and its comparison to international contexts. This analysis was driven to understand the potential replicability of our findings across different higher education institutions. This chapter encompasses SO1. The article that composes this chapter has been published in the Brazilian Journal of Computers in Education (RBIE)[6].

Chapter 3 serves as the centerpiece of this thesis. It presents the identification and initial assessments of MC^3, offering comprehensive descriptions of the methodologies employed throughout the study. The findings of this chapter shed light on the prevalent MC^3 that warrant attention in CS1 classrooms, while also elicit potential educational interventions to address these misconceptions. This chapter encompasses SO1 and SO2. The article that composes this chapter has also been published in the Brazilian Journal of Computers in Education (RBIE).

Chapter 4 further explores the identified MC^3 through a mixed-methods approach, comprising a large-scale study and a case study. The large-scale study aims to understand how students incorporate MC^3 into their code over the course of an academic term, as well as to explore the potential influence of classroom teaching contexts on students' incorporation of MC^3. Furthermore, the case study seeks to investigate the efficacy of explanation-driven MC^3 educational materials in aiding students' understanding and avoidance of these misconceptions. This chapters encompasses SO1, SO2, and SO3. This chapter was written as a journal article, but it has not been published yet.

Chapter 5 presents a discussion to connect the results obtained in previous chapters. The discussion is guided by an assessment of each one of the three Specific Objectives proposed in this thesis. Lastly, Chapter 6 provides the conclusions obtained in this thesis as well as elicits possible future work.

1.6 The Researcher-Participant-Instructor

Throughout the years of conducting this thesis, I have learned the significance of providing and analyzing context in any endeavor. With this in mind, I have decided to provide a brief overview of my background, believing that it will aid future readers in understanding the perspective from which this work arises.

As mentioned earlier in this chapter, my interest in educational research solidified during the final year of my bachelor's degree. At that time, I assessed the potential

[6] `https://sol.sbc.org.br/journals/index.php/rbie`

of educational software to enhance primary education, focusing particularly on Biology classes. This exploration extended into my master's, where I utilized visual programming blocks and agent-based modeling to facilitate teaching and learning in Biology across both primary and secondary educational levels.

The impetus for pursuing this thesis emerged during my tenure as a teaching assistant in a CS1 class during my master's. This hands-on experience allowed me to engage with students from diverse educational backgrounds, particularly those not enrolled in STEM programs. It is also an interesting fact that I did not have experience with autograders back then, although I knew about their usage. Following this experience, I found myself increasingly drawn to educational settings of this nature, prompting me to further investigate the field.

Finally, the title of this section encapsulates the perspective I envision about myself after having conducted the development of this thesis. It took time before I could fully engage in research with CS1 students, during which I also acted as their teaching assistant. In the final year of this thesis, I assumed the role of instructor for a CS1 class while I also was conducting my research. Reflecting on these experiences, it has been impossible to separate my participation in both research and teaching throughout this journey.

Chapter 2

A Syllabi Analysis of CS1 Courses from Brazilian Public Universities[1]

Abstract: The design and administration of methodological interventions is a possible way to address dropout and failure rates in undergraduate introductory programming courses (CS1). However, to implement such strategies, it is required to identify how CS1 courses are organized and offered to students. In this work, we analyzed 225 syllabi from CS1 courses distributed in 95 Brazilian public universities. We collected these syllabi from Brazilian undergraduate programs with focus on Computer Science aspects. We report context information regarding the most covered topics, most common CS1 course names, when undergraduates take the course, total class hours, and the programming paradigms and languages taught. The results indicate that the Brazilian scenario has its own characteristics that differs from those presented in related work conducted in other countries or world regions. In Brazil, we identified 90% of the analyzed CS1 courses teaches the procedural paradigm, with C as the most common programming language (53% of the total). These results differ from those conducted in other countries where Java and object-oriented are the most common languages and paradigm taught in CS1 courses. We believe that our results can be used to: (1) provide an update to those interested in the Brazilian scenario of CS1 courses; (2) support future interventions in teaching and learning of CS1; and (3) support the Brazilian community in the development of future CS1 syllabi.

2.1 Introduction

The utilization of computers, as a tool, continues to expand in most diverse areas. In 2005, it was estimated there would be over 90 million of end users in American workplaces, varying from spreadsheet users to actual programmers [110]. As consequence of this, more undergraduate programs are offering Computer Science (CS) classes to their students. These classes are often referred as *CS+X* in the literature [124]. Although CS+X classes are often in the Science, Technology, Engineering and Mathematics (STEM) field, such

[1]This paper was published in the Brazilian Journal of Computers in Education (RBIE). DOI: `https://doi.org/10.5753/rbie.2023.2870`

as Cognitive Psychology [25] and Biology [40], they can also be found in other areas, such as Law [123].

A typical course within the teachings of CS regards the concepts of computer programming. In 1978, the Association for Computing Machinery (ACM) defined the terms CS1 and CS2 to describe the first two programming courses a student takes in an CS undergraduate program [8]. These terms have been used over 40 years, typically assigning CS1 to basic concepts of programming, and CS2 to data structures. However, there is no consensus of what exactly should be taught in these courses [62]. Literature reviews also beckon this statement by mentioning that publications regarding CS1 course contents are still stable over the last few decades [14, 85]. In Brazil, CS1 courses have many different names, even within a same higher education institution [91]. Examples of the names include "Introduction to Programming", "Introduction to Algorithms", "Algorithms and Data Structures", "Algorithms and Computer Programming", among others.

Usually, undergraduate students start developing their logical and systematical thinking in CS1 courses. These courses also present a first programming language to the students. In other words, CS1 courses are an important background not only for the remainder of the CS undergraduate program, but also for the professional formation of the undergraduates [33, 125]. However, along with this importance, CS1 teaching faces recurrent challenges regarding high rates of failure and dropout [24, 76, 93, 98, 140]. To help solving these problems, several research with interventions in teaching and learning of CS1 have been made [7, 28, 29, 82, 135].

Given the importance and the challenges that CS1 courses have, how do we construct solutions that will work in the multiple scenarios in which teaching and learning of these courses happen? In terms of programming being hard to learn, Luxton-Reilly [84] said that we make introductory courses difficult by establishing unrealistic expectations upon novices. The same author also stated that revisiting what is expected from students at the end of introductory programming courses might be the key to improve students' learning, address negative impacts of disciplinary measures, and create a more equitable environment in these classes. A possible first step to revisit what is expected from students would be to understand which concepts are being taught in CS1 courses. This step proves to be another challenge because some educators might say that all introductory programming courses teach the same thing: the basics of computer science and computer programming [85]. However, if we are to follow this rationale, there would be no need for research about CS1 curricula - and that does not happen in practice.

This work was motivated by the potential that an assessment of the characteristics CS1 courses could have. Among these characteristics were the most covered topics. With this list of the frequency of the topics, researchers would be able to understand which topics are most or least covered so they could construct teaching and learning interventions. However, to ensure that this assessment would be holistic, the search ought to be done not only in terms of quantity, but also covering a broad geographic context. We decided to focus on Brazil because: (i) as far as we know, similar research in the literature do not state that Brazil was covered in their results [9, 12, 13, 63, 111]; and (ii) we did not find any other work that conducted a holistic search in Brazil to assess the characteristics of CS1 courses with focus on the most covered topics. Finally, we decided to limit our research

to Brazilian public universities, which are composed by federal, state, and municipal institutions. Public universities were selected because they are present in all geographical regions of the country, and they are the ones that most appear in international rankings, such as the World University Rankings of the Times Higher Education[2].

An interesting approach to help with the research of the most covered topics would be to consult entities that are directly involved with the design of undergraduate programs. In Brazil, the organization and research of CS teaching have always been conducted by the Ministry of Education together with the Brazilian Computer Society (SBC). Among the contributions by both groups are discussions that elaborate and assess CS undergraduate programs. The Formation Guidelines for Computer Science Undergraduate Programs [149] are a result of several research that culminated in orientation to develop pedagogical projects. The document is organized by the types of CS undergraduate program in Brazil. In a global scenario, the Computing Curricula 2020 [35] provides recommendations in the same way. ACM and the Institute of Electrical and Electronic Engineers (IEEE-CS) elaborated this document, and it is also endorsed by SBC.

The Formation Guidelines and the Computing Curricula 2020 are well suited documents for the creation of pedagogical projects. However, they focus on competencies that each student must develop by the end of the undergraduate program, not necessarily being achieved in each of the courses. Considering this, they would not be the ideal source to obtain information specifically for CS1 courses. In this work, we chose to search and analyze documents that had already been created by the public universities: their pedagogical project and the syllabi for CS1 courses. By analyzing these documents, we would be able not only to identify the covered topics, but also other characteristics of the CS1 courses.

We created five research questions to contextualize the Brazilian CS1 courses from public universities:

RQ1: *What are the most common topics covered in CS1 courses from Brazilian public universities?*

RQ2: *What are the most common CS1 courses' names from Brazilian public universities?*

RQ3: *When do Brazilian public universities' curricula suggest students take the CS1 course?*

RQ4: *What is the average of the total class hours of the CS1 courses from Brazilian public universities?*

RQ5: *What are the programming paradigms and programming languages taught in CS1 courses from Brazilian public universities?*

We manually searched the websites for 157 Brazilian public universities and curated 225 CS1 syllabi present in a total of 95 institutions. We used the pedagogical projects together with the CS1 syllabi to answer each of the aforementioned research questions. Our results indicate that the Brazilian scenario has characteristics that differs from those in other countries, such as the programming paradigm and languages that are taught.

We believe that our results can contribute to researchers interested in an update about the context of CS1 teaching in Brazil, especially in worldwide research where the spoken language can be a barrier (almost all Brazilian syllabi we found were only in Brazilian

[2] `https://www.timeshighereducation.com/world-university-rankings`

Portuguese, for example). The results can also be used to help in the construction of teaching and learning interventions of CS1 courses. Lastly, we believe that the list of most covered topics can also be used in the creation of new syllabi by Brazilian higher education institutes.

The remainder of this paper is organized as follows. In Section 2.2 we present the background and related work. In Section 2.3 we detail the methodology used, followed by the obtained results in Section 2.4. We discuss the obtained results in Section 2.5. In Section 2.6 we present the limitations and threats to validity of this research. Lastly, the conclusions are presented in Section 2.7.

2.2 Background and Related Work

The syllabus is a valuable tool in higher education because it is often the first formal way in which students receive information about a course. Syllabi analysis is important because the syllabus is an educational tool with functionalities that are commonly unknown to administration, faculty, and students [44]. While designing a syllabus or course outline, the instructor needs to take careful consideration of topics covered, assignments' due, and learning objectives [89]. McKeachie also says that, for undergraduates, the syllabus establishes expectations and directions for a particular course, thus providing a way of security. Even though the definitions of what a quality syllabus has not been clearly defined [44], some suggested models go way beyond the aforementioned characteristics. A course calendar, grading information with the rubrics that are used, additional resources, and a guide to use the syllabus are examples of suggested items to be included in a syllabus [57].

Becker and Fitzpatrick [13] analyzed syllabi from CS1 courses of 916 institutions present in the QS World University Rankings from 2016-2017[3]. The authors were motivated to answer what exactly CS1 teachers expect from their students at the end of the introductory course. While they were searching for learning outcomes, a total of 15 topics were among the most covered: testing and debugging, writing programs, and selection statements were the top three. Becker and Fitzpatrick also reported information about the most used programming languages, in which Java, Python, and C++ were the most used. Finally, they also created an online tool for the community in which it is possible to sort and analyze their gathered date about the syllabi. However, as mentioned in their work, they could only process English-created material. We believe this could possibly be the reason Brazil was not present in their analysis.

Porfirio, Pereira, and Maschio [101] also did an analysis of syllabi from CS1 courses. They consulted 10 Brazilian federal universities (two for each geographic region) listed in the RUF 2018 Ranking[4] for Computing Programs. The authors were interested in discovering basic concepts that every student should master independently of the adopted approach in CS1 courses. In their work, they reported 10 most covered topics, with conditional structures, repetition structures, and data types being the top three. The

[3] https://www.topuniversities.com/university-rankings/world-university-rankings/2016
[4] https://ruf.folha.uol.com.br/2018/ranking-de-cursos/computacao/

authors' main goal was to create an automated assessment of computer programming skills by analyzing source code, called the *A-Learn EvId* method.

Syllabi analysis is also present in other CS related areas. Fréchet, Savoine and Dufresne [52] analyzed syllabi of text-analysis courses from 45 graduate political science programs. The authors presented a systematic method for analyzing syllabi and retrieving information to help early-career professors and political science departments to build syllabi for text-analysis courses. The authors reported a method for evidencing most cited academic papers and books used in the syllabi, and about the choice of software between R or Python. Using the same systematic method, Abad, Ortiz-Holguin, and Boza [1] analyzed syllabi of 51 Distributed Systems courses to answer if what is being taught in these courses matches important curricula initiatives. The authors reported the most covered topics, books, and papers listed in the analyzed syllabi of Distributed Systems courses.

The aforementioned studies used syllabi analysis approaches to find common topics in CS1 and CS related areas, sometimes extending it to other information also presented in these documents. However, analyzing syllabi has been deemed a challenging task to do in a large scale [13, 130]. As a result of this, other ways of listing topics in CS1 were used in the literature, such as surveying academics [63, 111] and textbook analysis [18, 130]. In general, these studies did not have the most covered topics as a main objective, but instead focused on aspects such as importance or difficulty perceived of said topics. Hertz and Ford [63] culled a list with 17 topics from the literature and surveyed CS1 professors to investigate correlations between the importance of these topics and students' developed skills. Schulte and Bennedsen [111] did a similar approach in surveying professors to find what they teach, what they believe that should be taught, and the CS1 topics students tend to have difficulty with. Their survey used a list of 28 topics to compose the analysis.

Berges and Hubwieser [18] developed a semiautomated mechanism for textual analysis. The authors used this mechanism with five CS1 books, which addressed the object-oriented paradigm, to elaborate Concept Specification Maps. Their provided list of topics varied for each book, with their most populated listings ranging between 17 and 18 topics. Tew and Guzdial [130] used a bottom-up approach with 12 textbooks to identify concepts taught by multiple CS1 courses. Their initial analysis culminated in a wide list with more than 400 topics. However, after further refinements that used other established curricula, and sorting concepts by programming paradigm, they ended with a list of 29 topics. In the end, the authors used their result to create a validated assessment of CS1 topics known as Foundational CS1 (FCS1) [131].

We present a summary of the presented related work in Table 2.1. We briefly compare the methodology, total of respondents, syllabi or textbooks analyzed, and if each work covered Brazil in their analysis. We omitted research from Abad et al. [1] and Fréchet et al. [52] from the table because they did not evaluate CS1 courses.

Our work differs from the studies done by Becker and Fitzpatrick [13] and Porfirio et al. [101] because we analyzed 225 CS1 syllabi from Brazilian public universities, covering all geographical regions. We also report more information such as most common names of CS1 courses, research question also presented in Abad et al. [1] albeit their focus was on Distributed Systems courses. Our ranking of the most covered topics was done without filters of importance nor perceived difficulty of these topics, differing from other studies

Table 2.1: Summary of the related work presented in this section. Table is sorted alphabetically by the methodology.

Research	Methodology	N	Covered Brazil?
Hertz and Ford [63]	Online survey with instructors	99	Not mentioned*
Schulte and Bennedsen [111]	Online survey with instructors	349	Not mentioned*
Becker and Fitzpatrick [13]	Syllabi analysis	234	No
Porfirio et al. [101]	Syllabi analysis	10	Yes
Berges and Hubwieser [18]	Textbook analysis	5	N/A**
Tew and Guzdial [130]	Textbook analysis	12	N/A**
This work	Syllabi analysis	225	Yes

*Authors did not mention Brazil among the respondents' location.
**Verification not applicable since the methodology used textbooks.

in the field that identify difficult topics and develop methods to mitigate these difficulties [6, 7, 28, 95]. These kind of research shows where they succeeded or failed, thus enriching the community [136]. However, the field could benefit more from empirical applications involving the results obtained for the already developed methods instead of creating new ones [85].

2.3 Methods

In this work, we gathered data directly from publicly available sources provided from Brazilian universities to answer each proposed research question. As previously stated, we decided to focus on public universities because they are the most present in rankings, and they cover all Brazilian geographical regions, thus providing a broad criterion for analysis.

In Brazil, public universities are higher education institutions financially maintained by the government. As consequence of this, one of their main characteristics is to serve the public and collective interest [49]. Public universities are also known for their rigorous undergraduate admission process based on entrance exams, and the absence of tuition costs for the students. There are three types of public universities in Brazil: federal, state, and municipal. To guide our research in an ordered manner, we used three listings that described the universities from each desired group. According to the lists, there are 69 federal [144], 47 state [145], and 41 municipal [146]. Since many undergraduate programs can offer CS1 courses, we decided to limit our scope to Computer Science related undergraduate programs. As the Formation Guidelines [149] were organized by CS undergraduate programs, it seemed appropriate to use them in our work. We analyzed the following programs: Bachelor in Computer Science, Bachelor in Computer Science and Engineering, Bachelor in Software Engineering, and Bachelor in Information Systems. Another analyzed program can be described as a degree in Computer Science that prepares CS teachers to act in Brazilian's first and secondary educational levels. We did include undergraduate programs with minor difference in their names from the ones we sought, as well as programs which were in person or in distance learning format. It is important

to state that even though undergraduate programs that offers a Technology degree are present in the Formation Guidelines, we did not include them in our research.

2.3.1 Data Collection

The entire process of data collection was done manually since data had to be collected directly from publicly available sources from the institutions. The following steps describe how this process occurred for all types of public universities (federal, state, and municipal):

1. Selection of a public university from the corresponding base listing [144, 145, 146]. This base listing was composed of two types of institutions: universities and colleges. However, as our focus was Brazilian public universities, we discarded the latter type of institution from our analysis.

2. Navigation to the university's official website.

3. Search for the desired undergraduate programs offered by the university. We decided to verify all possible campi that had each targeted program so that we could identify distinct versions that CS1 programs may have. This step repeated itself until all desired undergraduate programs were analyzed. If the university did not have any of the targeted programs, it was discarded from our analysis.

4. We used two criteria for determining if a course was considered as CS1: whether it had focus on teaching computer programming concepts (including or not the teaching of a programming language); and whether it was the first course with this former criterion listed in the university's suggested curriculum order. Our approach was similar to Guo's [58] as we did not consider courses that only teach basic computer literacy. If more than one course had both criteria, we compared their syllabi and chose the one which had more topics covered. However, if a public university had two courses that taught different programming paradigms in the same curricular period, both courses were included. Also, if a CS1 course was divided into two, one taught in class and the other in the laboratory, we merged both.

5. Once a CS1 course had been identified, two documents were searched to answer the proposed research questions: the pedagogical project of the undergraduate courses, and the syllabus of the CS1 courses. We expected both documents to complement themselves regarding the information we needed, but our focus was the syllabi. However, these documents were not always sufficient to identify what we wanted. In some cases, the syllabi were already present in the pedagogical project, in others, though, they were not, leading us to search the institution's website for more documents. One example of other type of document was the program contents for a specific semester. If we could not find any other document within their website, we Google searched "<institution name> + <CS1 course name> + <syllabi>" (in Brazilian Portuguese) to find instructors' personal websites or repositories that contained information about the syllabi. If we could not find or did not have access to any syllabi information using the aforementioned methods (sometimes the official websites were offline or they required login information to have access), the

CS1 course was discarded from our analysis even if we had found other desired information about it.

6. Saving of the retrieved data by collecting and pasting directly from the sources.

At the end of the data collection process, we had one main document for each type of public university (federal, state, and municipal). The documents were organized by geographical region, institution, undergraduate program and CS1 course name. The data was collected in two cycles: the first happened during July and September of 2021, in which we focused on federal universities; and the second happened during April and July of 2022, in which we covered state and municipal universities.

2.3.2 Data Analysis

Since we used different approaches to answer the proposed research questions, we decided to describe the analysis methods for each one. We had identified two issues before our analysis began: the possibility of having multiple equivalent CS1 courses from the same institution in the assembled documents, and whether we would use or not the public universities' names in our results.

As we believe that having multiple equivalent CS1 courses would not add any impact in the results, we decided to maintain only one version of the equivalent courses, discarding the rest. This process only happened when the retrieved documents clearly stated that those courses were equivalent to each other. This also means that there was one syllabus per CS1 course. In the remainder of this article, we use these terms (*CS1 course* and *syllabi*) as synonyms when representing results. As for the other issue, we chose to present only the public university names in Appendix 1. The reason for this was because we did not have any intention of highlighting nor comparing these institutions in respect to how they structure their CS1 courses.

One final information we used in our general analysis was the specified year in which the data began to be *officially recognized* by the public university. We collected this information from each analyzed document e.g., the year in which the pedagogical project for the undergraduate program began. We decided to present this information to illustrate how old the collected data was. The emphasis in officially recognized means that if the university did have a more recent pedagogical project but it was still in the process of approval, we used the official one at that time. If we used more than one document to collect data about a CS1 course, we considered the year of the most recent one.

RQ1: *What are the most common topics covered in CS1 courses from Brazilian public universities?*

A CS education researcher analyzed the syllabi to identify the topics. For each found syllabus, he listed each of the covered topics in a worksheet. The list expanded with new topics as more syllabi were analyzed. During this analysis, the researcher used his experience to identify and group together topics with different names but meant the same concept. At the same time, the covered topics present in distinct syllabi were signaled and counted when the analysis ended.

Once the researcher had finished assembling the list with the topics and their frequencies, we sorted them decreasingly by the frequency. Since the goal was to identify the most common topics, we decided to report a subset of the initial list because the first assembled listing had many items. This subset was composed by applying two minimum thresholds regarding the frequencies. Finally, we decided to compare our listing with those assembled from related work. This comparison checked whether the topics were present or not in the other listings.

RQ2: *What are the most common CS1 courses' names from Brazilian public universities?*

We created a worksheet containing all the retrieved CS1 courses' names to answer this question. Even though courses that were divided into theoretical and practical classes were merged, we decided to keep only one name in the analysis (generally being the course regarding the theoretical class). The reason for that was because in most cases in which this situation occurred, the names were, for example, "Algorithms" and "Laboratory of Algorithms". If a course had numbered and not numbered names, we chose to consider both e.g., "Algorithms" and "Algorithms 1". This decision was made because this occurrence means that some universities have subsequent courses with the same name.

RQ3: *When do Brazilian public universities' curricula suggest students take the CS1 course?*

To answer this question, we analyzed how the public universities divide their curricula, as some institutions organize them by semester (semiannual), and others, by years (annual). Then we grouped together when their CS1 courses were suggested for the students to take e.g., first semester or first year. There were some cases in which even though the institution divided their curricula in years, the CS1 course only happens in one of its semesters. When this happened, the course was considered as semiannual.

RQ4: *What is the average of the total class hours of the CS1 courses from Brazilian public universities?*

Since not all CS1 courses explicitly stated how their class hours are divided (e.g., theory and practice), we chose to consider the absolute total. In other words, even if a specific CS1 course stated how its class hours are divided, the hours were summed. The same rule applied in cases when one CS1 course was divided into theoretical and practical classes. Before computing the average, we grouped courses with similar length. The reason for that was because it would not be reasonable to aggregate the total class hours for semiannual and annual courses. CS1 courses that happened in one semester of a scholarly year were considered as semiannual for this analysis. Finally, it is important to say that this data was not available for all found CS1 courses.

RQ5: *What are the programming paradigms and programming languages taught in CS1 courses from Brazilian public universities?*

The syllabi from the CS1 courses were used to answer this question. However, the retrieved syllabi did not always explicitly state the paradigm or the language. When this happened, the researcher responsible for the analysis decided to infer it from both the

syllabus description and the suggested bibliography. While the paradigm could also be inferred by the programming language (e.g., Haskell would imply the functional paradigm), most object-oriented languages can be used to teach the procedural paradigm or the object-oriented paradigm (know together as the imperative paradigm) [85]. Based on this, the researcher classified the paradigm based on his experience regarding the topics covered in the CS1 course. It was reported, inferred or not, a programming paradigm for every syllabus analyzed.

As for the programming language, the researcher consulted the suggested books in the bibliography (both basic and recommended) in the order they were listed. Then, he used the first one in which it was possible to infer a programming language. It is important to say that not all retrieved syllabi included a bibliography, and some books did not specify or used more than one programming language. In these cases, the language was classified as not possible to infer. We report the percentage of each inferred paradigm and language in our results in Section 2.4.

2.4 Results

In this section, we describe the results obtained with the execution of the data collection and analysis detailed in Section 2.3. We begin by presenting the general information about the public universities and the retrieved documents. Then we report the results for each research question, focusing on details about the application of the proposed methods. We present the discussion about the results and the answers for each research question in Section 2.5.

2.4.1 General Information

Table 2.2 presents the total amount of public universities and CS1 syllabi analyzed in this research. The associated type of public institution (federal, state, and municipal) is also described. For the public universities, we report the total that was present in the base listings [144, 145, 146], the total universities that offered a targeted CS related undergraduate program for this research (detailed in Section 2.3), and the total universities that we used to compose our main results. For the CS1 syllabi, we report the total syllabi that we retrieved from the analyzed documents, and the distinct syllabi that we used to compose our main results. In total, we used 95 public universities and 225 CS1 syllabi in our main results.

Table 2.3 presents the geographic distribution of public universities and CS1 syllabi. We decided to report this distribution only for the universities and syllabi that composed our main results. In other words, Table 2.3 expands the numbers reported on both *Used* columns from Table 2.2. Our results managed to provide a broad analysis in terms of covering all Brazilian geographic regions in this research. This was only possible because of the federal universities since, after the discarding process mentioned in Section 2.3, state and municipal institutions were not present in all regions.

The distribution of the years of the documents in which we retrieved the syllabi information from is represented in Figure 2.1. As explained in Section 2.3, we only used the

Table 2.2: General distribution of the public universities and CS1 syllabi analyzed.

	Universities			**CS1 Syllabi**	
Type of institution	Present in Base Listings	Had Targeted CS Programs	Used	Retrieved	Used
Federal	69	63	61	195	150
State	47	35	32	88	72
Municipal	41	15	2	8	3
Total	157	113	**95**	291	**225**

Table 2.3: Brazilian geographical distribution of the public universities and CS1 syllabi that composed the main results of this research

	Universities			**CS1 Syllabi**		
Region	Federal	State	Municipal	Federal	State	Municipal
Center-west	8	3	0	17	10	0
North	9	0	0	22	0	0
Northeast	17	12	0	48	20	0
South	9	8	1	28	17	1
Southeast	18	9	1	35	25	2
Total	61	32	2	150	72	3

most recent documents that were officially recognized by the university. We assessed this factor even when we had to use other sources (such as class slides retrieved from instructors' websites) by checking if their publication year was within the official pedagogical project from the undergraduate program of the institution at that time. We present the distribution for each type of public university (federal, state, and municipal). The asterisk in 2022 indicates that the process of data collection and analysis happened between 2021 and 2022, meaning that the retrieved documents in 2022 could not be valid for the whole year. The numbers do not add up to 225 because we could not identify a year for 8 syllabi. While there were documents dating from as far as 2006, more than half (116) were from 2018 to 2022.

2.4.2 RQ1: CS1 Topics

As described in Section 2.3, we identified different CS1 topics by manually reading all the 225 syllabi from CS1 courses that were not considered equivalent to each other. We organized the initial listing by counting how many syllabi each topic covered, then we sorted the list decreasingly. In total, 72 different topics were identified. Although we decided to omit this initial list from our reports, we illustrate the frequency distribution of the 72 identified CS1 topics in Figure 2.2. The analysis of this distribution indicated that 51 different topics were present in less than 25 common syllabi. In other words, this means that approximately 71% of the identified topics were common to less than 11% of the total syllabi used in this research. On the other hand, there was no topic common to all 225 syllabi. We discuss the possible causes of both factors in Section 2.5.

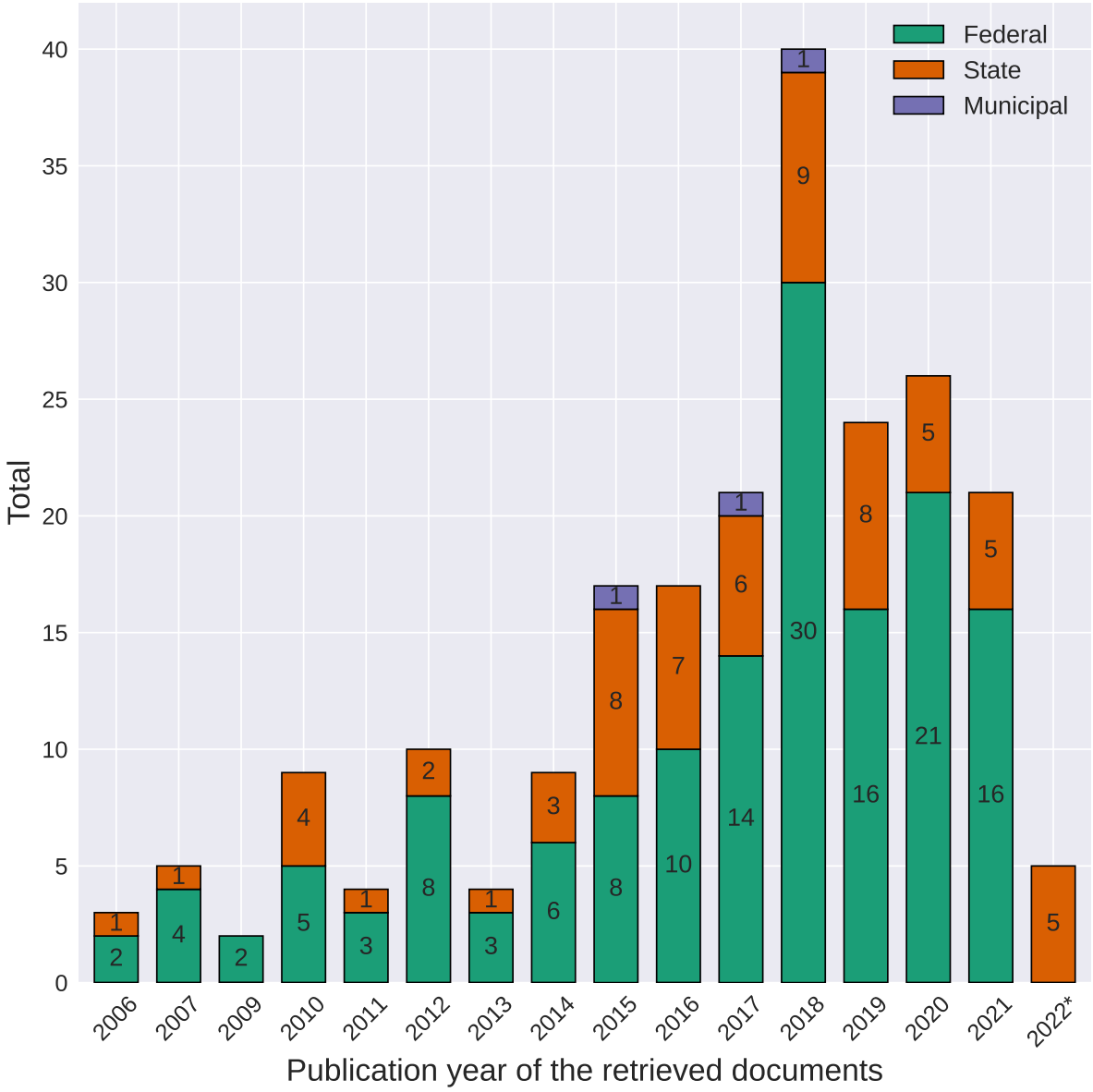

Figure 2.1: Distribution of the publication year of the retrieved documents that contained syllabi information. N = 217. It was not possible to retrieve the year in 8 syllabi. The asterisk in 2022 indicates that the corresponding retrieved syllabi might not be valid for the whole year since the data collection happened between 2021 and 2022.

Table 2.4 presents the ranking of the most covered topics. Since we were interested in the most common (RQ1), we decided to consider a subset of the initial listing with 72 topics. To do that, two thresholds were applied based upon the total of 225 syllabi used: the first was 10%, and the second was 33%. In total, 21 topics remained after applying the first threshold, and after the second, 12 topics remained. For each topic we report: the descriptive name, in which we tried to describe the different ways each syllabus referred to a same topic; the total number of the syllabi that each topic had in common; and, as a complement of the latter, the fraction of its frequency in terms of the 225 syllabi used. Table 2.4 is already represented with the initial threshold i.e., all the 21 topics appeared

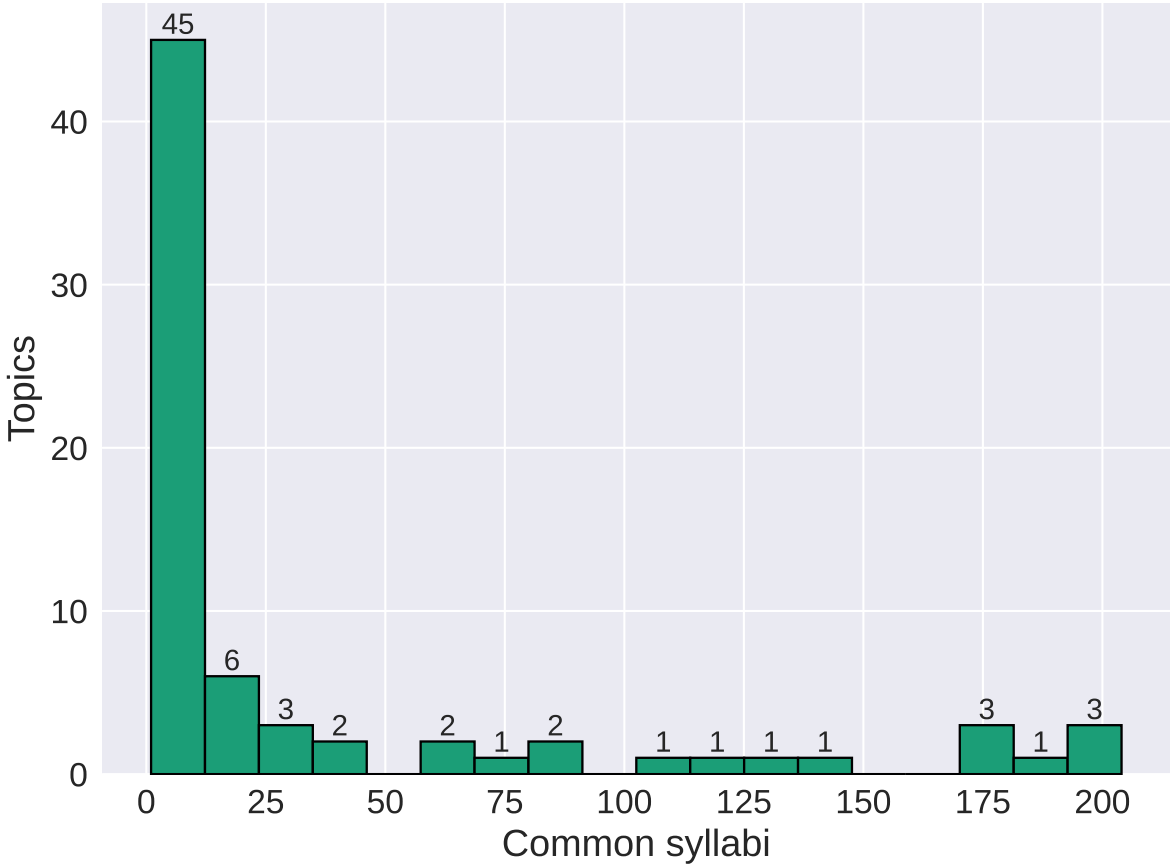

Figure 2.2: Frequency distribution of the presence of the initial 72 identified CS1 topics.

in at least 33% of total syllabi. A horizontal line defines the second threshold: the first 12 topics appeared in at least 10% of total syllabi.

The comparison of our listing with related work is also present in Table 2.4. We used the listings retrieved from Becker and Fitzpatrick [13] and Porfirio et al. [101] because they also analyzed CS1 syllabi. We also included the listings from Hertz and Ford [63], and Schulte and Bennedsen [111] to the comparison. It is important to notice that we only considered the 15 most covered topics from Becker and Fitzpatrick [13] since the authors explicitly highlighted them. As explained in Section 2.3, we compared if a topic in our listing was present or not in the others, marking the presence with the ✓symbol. An example reading of Table 2.4 is that the topic *Conditional commands* appeared in 204 syllabi, and was common to 91% of all 225 syllabi. The same topic is also present in Becker and Fitzpatrick [13], Porfirio et al. [101], Hertz and Ford [63], and Schulte and Bennedsen [111].

2.4.3 RQ2: CS1 Courses' Names

We collected the CS1 course name directly from the used syllabi to calculate the results. The methodology was analogous to the one used in finding the most covered topics: we counted the frequency of each course name and ranked their total decreasingly. Table 2.5 presents the listing with the most common names. The total does not add up to 225

Table 2.4: Ranking of the most covered CS1 topics from Brazilian public universities. The horizontal line highlights the second threshold applied. Becker, Porfirio, Hertz, Schulte represent listings retrieved from [13, 63, 101, 111], respectively.

Topic	Total	%	Becker	Porfirio	Hertz	Schulte
Conditional commands	204	91	✓	✓	✓	✓
Variables, constants, and assignments	201	89	✓	✓	✓	✓
Repetition commands	200	89	✓	✓	✓	✓
One-dimensional homogeneous composite variables	184	82	✓	✓	✓	✓
Arithmetical, logical, and relational expressions	181	80	✓	✓	✓	
Functions, modularization, subprograms	180	80	✓	✓	✓	✓
Multidimensional homogeneous composite variables	173	77	✓	✓		✓
Data input/output	141	63	✓	✓	✓	
Algorithm representation forms	127	56	✓			✓
Heterogeneous composite variables	118	52	✓			✓
Recursion	85	38	✓		✓	✓
Scope of variables and parameter usage	82	36			✓	✓
File handling	69	31	✓		✓	
Basic computer organization	64	28				
Pointers and dynamic memory allocation	61	27				✓
Debugging	40	18	✓		✓	✓
Documentation	37	16	✓			
Testing	33	15	✓		✓	
Sorting algorithms	30	13			✓	
Search algorithms	30	13				
Programming environments	22	10				✓

because we decided to omit names that appeared less than 5 times. It is important to notice that the translation of the course names was conducted by ourselves. This means that the names might not be exactly how each public university would translate them officially.

Table 2.5: Most common CS1 courses' names from Brazilian public universities.

Name (original)	Name (our translation)	Total
Programação 1	Programming 1	23
Introdução à Programação	Introduction to Programming	20
Algoritmos e Programação 1	Algorithms and Programming 1	16
Algoritmos e Estruturas de Dados 1	Algorithms and Data Structures 1	16
Algoritmos	Algorithms	15
Algoritmos e Programação	Algorithms and Programming	14
Algoritmos 1	Algorithms 1	11
Fundamentos de Programação	Programming Fundamentals	10
Introdução à Computação	Introduction to Computing	6
Técnicas de Programação 1	Programming Techniques 1	5
Programação de Computadores 1	Computer Programming 1	5
Algoritmos e Programação de Computadores	Algorithms and Computer Programming	5

2.4.4 RQ3: When Students Take the CS1 Course

We mostly used the pedagogical projects from each undergraduate program to identify when the public universities suggests that the students take the CS1 course. As explained

in Section 2.3, each course was separated in terms of their duration and how each public university divided their curricula. Table 2.6 presents the results obtained in this analysis, sorted by the total number of CS1 courses for each suggested period. The *quarter* suggested period is composed of a four-month cycle, meaning that the institution divided its scholarly year in three quarters.

Table 2.6: Periods in which Brazilian public universities suggests that students should take the CS1 course.

Suggested Period	**CS1 Courses**
1st Semester	194
2nd Semester	17
1st Year	10
2nd Year	2
1st Quarter	1
4th Semester	1
Total	225

2.4.5 RQ4: Class Hours Duration

Since most of the CS1 courses happened in an annual or semiannual format, we decided to group them by this category independently of when the CS1 course is suggested in the curriculum. This means that we considered courses in the 1st or 2nd Year as *annual*, and courses in the 1st, 2nd, and 4th Semester as *semiannual* (these items are from Table 2.6).

Figure 2.3 illustrates the distribution of total class hours (combining theory and practice) for both semiannual and annual CS1 courses. An example analysis of the figure reveals that 108 semiannual CS1 courses had a total class duration between 60 and 75 hours, while only 1 annual course fell within the same interval. On average, semiannual courses had a total class duration of 79 hours, with a standard deviation of 21. In contrast, annual courses had an average total class duration of 146 hours, with a standard deviation of 54. It should be noted that class hours for five courses, including the one classified in the 1st Quarter (Table 2.6), could not be found.

2.4.6 RQ5: Programming Paradigms and Languages

As mentioned in Section 2.3, the syllabi were used to identify the programming paradigms and languages taught in the CS1 courses. However, this information was not often explicitly stated. Because of that, we decided to infer it from the covered topics and the bibliography listed in the syllabi. Table 2.7 presents the programming paradigms taught in CS1 courses, sorted decreasingly by the total number of courses. Table 2.8 lists the programming languages, also sorted decreasingly by the total number of courses. For both tables, we also inform the approximate percentage of courses in which we had to infer the information. This means that, for example, in Table 2.7, from the total of 202 courses identified to teach the procedural paradigm, 92 (46%) were inferred from the syllabi (using the covered topics). Another example, in Table 2.8, from the total of 120 courses that

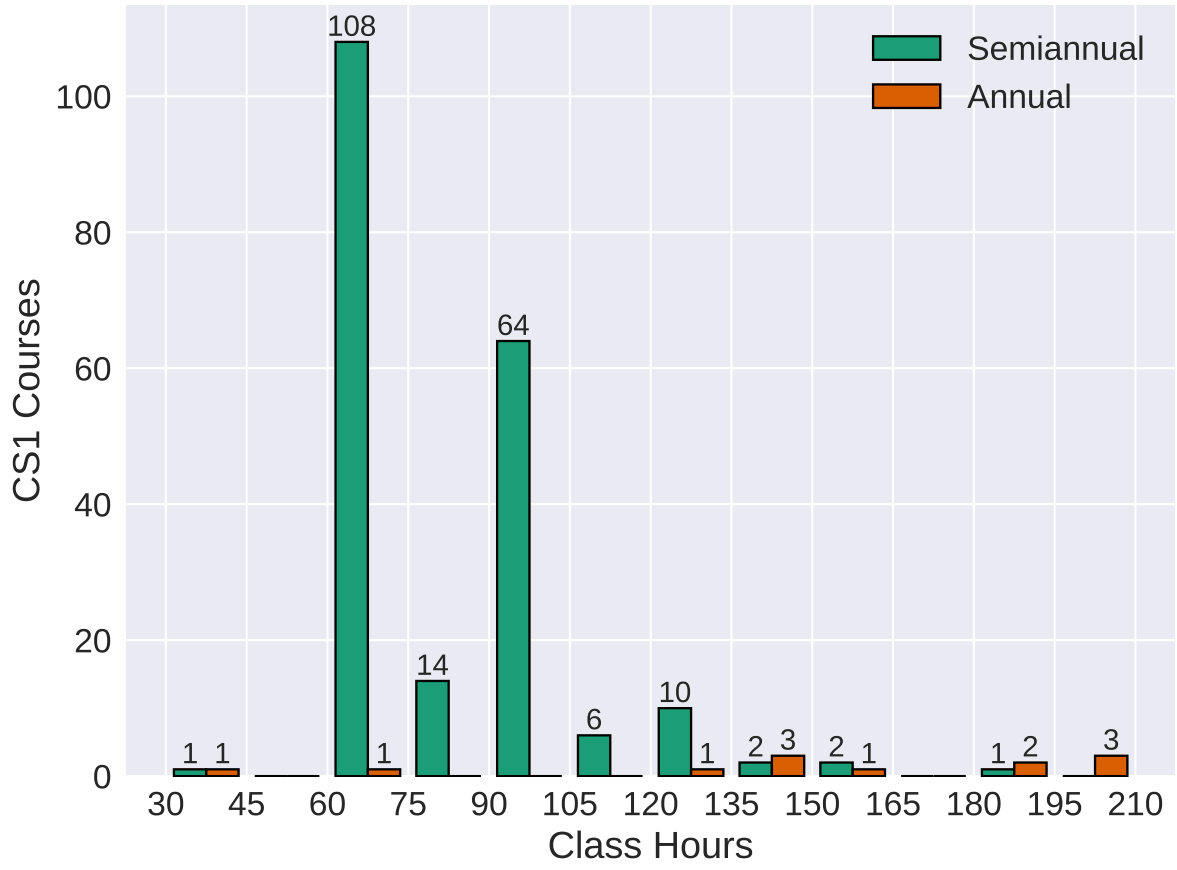

Figure 2.3: Total class hours distribution. N = 220. It was not possible to retrieve the class hours in 5 syllabi.

taught the C programming language, 86 (72%) were inferred from the syllabi (using the first listed item in the bibliography in which it was possible to infer a programming language). On a final note, the CS1 courses from Table 2.7 do not add up to 225 because we omitted one CS1 course that taught computational thinking concepts. The CS1 courses from Table 2.8 also do not add up to 225 because we could not infer the programming language of 57 syllabi.

Table 2.7: Programming paradigms taught in CS1 courses from Brazilian public universities.

Paradigm	**CS1 Courses**	**Total (%)***	**Inferred (%)****
Procedural	202	90	46
Object-oriented	16	7	0
Functional	6	3	33

* % of all 225 CS1 courses.
** % of the corresponding number of CS1 Courses.

Table 2.8: Programming languages taught in CS1 courses from Brazilian public universities.

Language	CS1 Courses	Total (%)*	Inferred (%)**
C	120	53	72
Python	19	8	68
Java	10	4	80
Pascal	7	3	71
C++	6	3	83
Haskell	5	2	60
Scratch	1	<1	100

* % of all 225 CS1 courses.
** % of the corresponding number of CS1 Courses.

2.5 Discussion

We present the discussion of the obtained results in this section. We begin by discussing the assets used in this work (public universities and the syllabi), then we follow by the contextualization of the CS1 courses from public Brazilian universities (RQ2-RQ5), and finally we address implications regarding the most covered topics in these courses (RQ1).

2.5.1 Brazilian Public Universities

The distribution of public universities presented on Table 2.2 indicates that there are more federal institutions than state and municipal, although these last two have similar numbers. However, it becomes clear that not all institutions offer the CS undergraduate programs that we were interested in. Specifically in terms of the municipal institutions, approximately only one third of the total offers these CS undergraduate programs. This could be explained by the fact that there are municipals dedicated to specific areas such as Medicine, Humanities or Law: these institutions did not offer any CS undergraduate program. The same fact could also be applied to state and federal universities, albeit in a lower rate of occurrence. There is also the fact of the presence of programs that offers a Technology degree: they were sometimes present in institutions, but we did not analyze them. We also discarded CS1 courses that we were not able to retrieve the syllabi, thus, if this occurred for all CS1 courses from an institution, the whole institution was discarded. While the number of federal and state universities did not vary much between the ones that had a CS undergraduate program of interest and the ones used, these totals varied for municipal institutions. In fact, we had difficulties in finding the syllabi for the CS1 courses present in the municipal universities.

As mentioned in Section 2.4, the geographic distribution of the public universities used in this research (Table 2.3) covered all Brazilian geographic regions albeit it was only possible because of the federal institutions. Northeast, southeast, and south are the regions with most universities and syllabus for CS1 courses used in this research. We believe that the reason for this was the presence of different CS1 courses in different campi from the same institution that were not equivalent to each other. These regions are the most populated in Brazil. Porfirio et al. [101] mentioned to have retrieved ten syllabi

from ten federal public universities: two for each Brazilian geographic region. Since we managed to analyze 225 syllabi, our results can be seen as an update to theirs.

The year distribution of the retrieved documents (Figure 2.1) concentrates more than half in more recent years, with 2018 being most frequent year. This result could indicate that these documents might be close to what is being currently taught in the CS1 classes from Brazilian public universities. However, as explained before, we did not consider pedagogical projects still under approval at that time. This means that an updated version of the CS1 could be in effect by the time of the publication of this research if the new pedagogical project was approved in the meantime. The documents retrieved in the years 2020 and 2021 are also important to consider. As consequence of the *Sars-Cov-2* pandemic, many Brazilian institutions created Emergency School Periods to be able to teach. The adaptation necessary for the implementation of these periods could have impacts on the concepts taught in CS1 classes, especially for those universities in which the in person learning format was predominant. While Becker and Fitzpatrick [13] and Porfirio et al. [101] do not directly inform about the years of their retrieved syllabi, all authors used universities from rankings created between 2016 and 2018. Since our results go from 2006 to 2022, they can be seen as a complement of the research of these authors.

2.5.2 CS1 Syllabi

The syllabi used in this search were not homogeneous, containing distinct levels of detail. We found syllabi that were defined in few lines and others that detailed the topics taught per week. We also identified that syllabi used different ways to express the same topic e.g., loops were sometimes called "repetition commands", "iterative commands" or even represented as "control structures" (combined with conditional commands). As explained in Sections 2.3 and 2.4, this was the reason a CS education researcher had to use his experience to identify equivalent topics represented with different names. We illustrate the heterogeneity of the syllabi by presenting examples from different public universities below, classified as S1 and S2. It is important to remember that these syllabi were originally elaborated in Brazilian Portuguese and were translated by the authors.

S1: Algorithms, fundamental programming concepts, expressions, control flow, functions and procedures, pointers, vectors and matrices, strings, dynamic allocation, structured types, files.

S2: Basics of programming logic: algorithms characteristics, algorithm representations, programs, instruction, sequences, successive refinement. Concepts of a procedural language: the C programming language, compiler, basic data types, constants and variables, comments, reserved words, logic and arithmetic expressions, assignment commands, data input and output. Control structures: conditional structures, iterative structures. Structured data types: vectors, matrices, dynamic memory allocation, pointers, user defined types. Modularization: functions, scope, parameters, recursion, file handling.

Porfirio et al. [101] mentioned that syllabi may use different ways to express the same topic. In S1, we considered that *control flow* is related to both conditional and iterative commands, topic explicitly stated as *control structures* in S2. Becker and Fitzpatrick [13] also reported this factor in a similar way since the authors presented a concept frequency

in terms of being explicit or not explicit in the analyzed syllabi. In our analysis, we found topics with vague meanings e.g., *fundamental programming concepts* (present in S2) could have different meanings depending on the programming paradigm.

We did not find any CS1 syllabi that was described with the detail levels that Grunert [57] specified. At best, we encountered the total class hours dedicated to each covered topic. In fact, most descriptions of the syllabi could be classified as course outlines. However, as McKeachie [89] stated that the instructor needs to take careful considerations regarding the covered topics, assignments' due, and learning objectives while constructing both syllabus and course outline, we did not discard any documents regarding its level of details. The data collection process used in this research was challenging, since we had to search the universities websites. If there was a national repository of syllabi, future research like our work would be simpler because researchers would be able to retrieve data by applying searching techniques like those used in systematic reviews, thus dedicating their focus on the assessment of topics. There are initiatives of repositories with these characteristics [64, 133].

We do not believe that the absence of a topic in a syllabus implies that the related CS1 course does not cover it. S1, for example, does not explicitly state variables and constants, but they state vectors, matrices, and strings. This means that the presence of advanced topics that require basic ones implies that the latter topics are covered in the course but were omitted in the syllabus. This could explain why no topic appeared in all 225 syllabi (Figure 2.2 and Table 2.4). On the other hand, the presence of many different topics in few syllabi (Figure 2.2) could be explained by topics covered by other programming paradigms (such as the functional, which only appeared in 6 out of the 225 syllabi (Table 2.7)). The autonomy Brazilian public universities have could also have influenced the topic distribution, because institutions could be teaching specific topics to prepare professionals with a particular set of skills required for that geographic region, or to balance students with different background.

2.5.3 Contextualization of Brazilian CS1 Courses

Based on the results obtained from RQ3 (Table 2.6), almost 99% of the analyzed CS1 courses are offered within the first scholarly year of the undergraduate programs. This was expected since most Brazilian universities divide their curricula in semiannual periods. This result also means that students are exposed to CS1 concepts early in their academic life. The only exception for this is one course that is offered in the fourth semester. We identified that this CS1 course is from a Bachelor in Computer Science and Engineering. While we could not find any reasons for this occurrence, the CS1 course is present at the end of the basic cycle. The basic cycle is a biannual period that aggregates common courses between Brazilian Bachelor in Engineering undergraduate programs. RQ4 expresses that annual CS1 courses have an average total class hours doubled than semiannual ones. This factor indicates consistency among these averages even though annual CS1 courses are not common when compared to semiannual courses identified in this research.

As explained in Section 2.3, we considered numbered entries as individual names to answer RQ2 (Table 2.5). The presence of these numbered entries in our results could indicate that there are other programming courses in sequence. In fact, we identified courses like that while searching for the introductory programming courses: in these cases, the CS1 course was a prerequisite to the subsequent one. This indicates that some Brazilian public universities organize their curricula by providing CS1 and CS2 courses although we only focused on the first one. We exemplify this factor by the presence of 16 CS1 courses named *Algorithms and Data Structures 1*. While data structures can be seen as a CS2 topic, some universities might divide its topics between CS1 and CS2, thus corroborating that there is no consensus among these courses [62]. Examples of course names that were omitted in Table 2.5 (because they were present in less than five courses) were: variations from the listed courses (e.g., Programming and Data Structures 1, Programming Principles); names containing the taught programming paradigm (e.g., Functional Programming, Introduction to Structured Programming, Object-oriented Programming Language); and other specific names (e.g., Applied Informatics, Information Processing).

Table 2.7 shows the programming paradigm of all used syllabi. There is a major gap between procedural (202 syllabi) and the object-oriented paradigm (16 syllabi), and this gap is increased when compared to the functional paradigm (6 syllabi). Using our criteria to identify CS1 courses (Section 2.3), we identified one course that taught computational thinking [147] concepts. Since it might not be fit to classify it as a programming paradigm, we decided to omit it from the table, thus the total does not add up to 225. Our inferring methodology was the same for all syllabi, used only when there was no explicit word or topic containing the paradigm. It is important to notice that we did not need to infer any course that teaches the object-oriented paradigm because it was explicitly stated in all 16 corresponding syllabi.

The other result that complements the answer to RQ5 is the programming language distribution (Table 2.8). Although C, Python, and Java are the most common languages, there is a major gap between C and the rest. C is more than 6 times higher than Python (119 compared to 19, respectively). This result differs from others found in the literature, which has reports indicating that Java is the predominant language [12, 13, 112]. We identified some reasons for this. First, our results are limited to Brazilian public universities. We did not find any related work that explicitly said to cover Brazil, but there are research that focused on other specific countries. Avouris [9] analyzed 121 CS1 courses from Greece and C was the most used language, appearing in 37 courses. Becker [12] surveyed instructors from 39 CS1 courses and Java was the most used language, appearing in 49% of the courses. Second, the methodology used could also have impacted in the results. Research that surveyed instructors directly are more precise than those who used syllabi (like this research) that might be outdated. The third reason is a direct consequence of the second because we retrieved information from documents published from 2006 and 2022, even though most of them were from 2018 and further. Lastly, the publication year of the used documents could also have impacted in our inferring methodology, since the bibliography used to infer the programming language could also be outdated.

To compare if there was any difference when limiting to recent years, we filtered the results from Table 2.8 to include only CS1 courses in which we obtained information from 2018 to 2022. The results obtained from this filtering is presented in Table 2.9. There were 124 syllabi from 2018 to 2022, but it was not possible to infer a language from 28 of them. In general, the results from the filtering do not differ much from our initial findings. The gap between C and Python was not altered (45% of the total courses before and after). Pascal and C++ also appeared less times (both were 3% of the total courses before, and 2% and 1% after, respectively).

We conclude that these results indicate that the Brazilian scenario has its own contexts for teaching CS1, differing from those in other countries or world regions. In general, all the obtained information about the contextualization of CS1 courses is a consequence of the aforementioned autonomy that Brazilian public universities have. The majority of CS1 courses teach the procedural paradigm, perhaps due to a general consensus that it is the best approach for beginner programmers. Regarding the programming language, there may be several factors that contribute to C being the most taught. One factor could be instructors' traditionalist view about the language, preferring it over others, arguably. However, there may also be factors related to the costs associated with training faculty to teach other programming languages, such as Python. We make this statement because the ability to program in a language and the ability to effectively teach it are two distinct skills.

Table 2.9: Comparison of the programming languages identified from documents ranging from 2006-2022 to those obtained from 2018-2022. It was possible to infer a programming language from 168 syllabi in the 2006-2022 period, and from 96 syllabi in the 2018-2022 period.

	2006-2022			**2018-2022**		
Language	**CS1 Courses**	**Total (%)***	**Inferred (%)*****	**CS1 Courses**	**Total (%)****	**Inferred (%)**
C	120	53	72	70	56	67
Python	19	8	68	14	11	64
Java	10	4	80	8	6	75
Pascal	7	3	71	2	2	0
C++	6	3	83	1	1	100
Haskell	5	2	60	1	1	0
Scratch	1	<1	100	0	0	-

* % of 225 CS1 courses present in the 2006-2022 period.
** % of 124 CS1 courses present in the 2018-2022 period.
*** % of the corresponding number of CS1 Courses.

2.5.4 Covered Topics

As detailed in Section 2.4, we omitted the full list with 72 topics because we wanted to identify the most covered topics in RQ1 (Table 2.4). Our decision to create two thresholds (common topics present in 10% and 33% of the total syllabi, respectively) was established to highlight the topics that were most common. Topics that remained outside the thresholds (i.e., appeared in less than 33% of total syllabi) include, but are not limited to: object-oriented and functional paradigm concepts (e.g., classes, objects, monads, func-

tion overloading); basic computer usage (word editors, spreadsheets, operating system, Internet browser); and advanced algorithms (geometric, non-deterministic). The absence of some of these topics in the thresholds corroborate with the procedural paradigm being the one most taught. As stated earlier, universities have the autonomy to select topics based on the desired professional outcome.

The 12 topics listed in the second threshold (Table 2.4) represent concepts present in the procedural paradigm. A special note is that the topic *Algorithm representation forms* was identified in syllabi that did not start with a specific programming language, teaching the other concepts via pseudocode. We noted CS1 courses that only begins teaching a programming language at the end of the course, while others did not mention any language. We conclude that this also contributed to the number of courses in which we could not infer the programming language. We also decided to list a group of concepts in a specific topic because of the different ways syllabi referred to them. Some examples of topics were: *One-dimensional homogeneous composite variables*, which included topics as vectors, lists, arrays, and strings; *Multidimensional homogeneous composite variables*, which included matrices; and *Heterogeneous composite variables*, which contained structures, unions, and dictionaries.

We also noted discrepancies regarding the frequency of topics that are closely related to each other. For example, in Table 2.4, *Functions, modularizations, and subprograms* appear in 80% of all syllabi whereas *Scope of variables and parameter usage* and *Recursion* appear in 36% and 38%, respectively. There could be a couple reasons for this, and they depend on the programming paradigm that each course teaches. *Scope of variables and parameter usage* could just have been omitted in the syllabus since it already had listed *Functions, modularizations, and subprograms*. However, since there were syllabi reporting both, we decided to treat them as separate topics. *Recursion*, on the other hand, could be related to the elaboration of the CS1 course itself. Since recursion is often considered a difficult topic to teach and learn [28], some curricula might delay its teaching to subsequent programming courses. Another relatable example regards the gaps between composite variables: there is one gap between one and multidimensional homogeneous, and an even greater gap with the heterogeneous types.

2.5.4.1 Comparison with Topics Listed in Related Work

The 12 topics present in the second threshold (at least 33% of total syllabi) appears in at least 2 out of the 4 analyzed related work (Table 2.4). This is an important factor because it establishes a certain degree of similarity with listings obtained from related work even though they were conducted in a different context (except for Porfirio et al. [101]).

We identified that some topics which were classified as distinct in Table 2.4 were sometimes grouped together in related work. For instance, in our work, we reported *Conditional commands* and *Repetition commands* as distinct topics. Becker and Fitzpatrick [13] listed *Selective statements (if/else/etc.)* separately from *Repetition & loops*, Porfirio et al. [101] listed *Conditional structures* separately from *Repetition structures*, Hertz and Ford [63] grouped these concepts as *Control constructs*, and Schulte and Bennedsen [111] grouped them as *Selection and Iteration*. Another example is *Variables, constants, and as-*

signments and *Arithmetical, logical, and relational expressions* were grouped as *Variables, types, expressions* [63] and as *Variables, assignment, arithmetic operators, declarations, data types* [13]. Since all reported listings were built upon the authors' experience, including our work, we consider this difference in grouping as just a matter of opinion in reporting the topics.

Our listing contemplated 9 of the 10 topics listed in Porfirio et al. [101]. Table 2.4 shows only 8 because they listed *Data types* as a separate topic, and we considered it within *Variables, constants, and assignments*. Another topic presented in this related work was *Introduction to programming*. This was an abstract concept in which we did not consider as a CS1 topic per se, thus we did not include it in our listing. Since Porfirio et al. [101] assessed a subset of 10 Brazilian federal universities, we consider our listing as an expansion of their initial work regarding the Brazilian scenario.

Debugging and *Testing* are topics that were not present in our second threshold but appeared in 75% and 50% of the related work, respectively. *Debugging* appeared in 40 out of 225 syllabi (18%). This concept is related to the teaching of specific tools used to debug code and some public universities might not teach them in the introductory programming course. Similarly, *Testing* appeared in 33 out of 225 syllabi (15%). Specific teachings of this topic could be covered in future courses that focuses more on software engineering concepts. The same analogy could also be applied to *Documentation*.

Lastly, we also searched for topics that were reported in the related work but not in our listing. Examples of topics in this classification were: abstract concepts (e.g., problem solving, writing programs, mental models); object-oriented concepts (e.g., classes and objects, inheritance and polymorphism, encapsulation); advanced data structures (e.g., graphs, trees, linked-lists); and algorithm efficiency (e.g., big-O notation).

The comparison of our listing with those obtained from related work makes the identification of common topics possible. However, except for Porfirio et al. [101], all research were conducted regarding different contexts, especially nationwide. This is corroborated by the results obtained from RQ5 (Tables 2.7 and 2.8). Since the programming paradigm taught in CS1 courses from Brazilian public universities (procedural was the most common) is different from the one taught in other countries (object-oriented), topic listings would certainly be different. All of this contributes to Hertz's [62] view that there is no consensus of what is taught in CS1 and CS2.

2.5.4.2 Grouping of the Second Threshold

As consequence of the discussion this section provided, the second threshold applied in Table 2.4 answers RQ1, our main research question. We decided to group the 12 highlighted topics by joining the closely related, using data from the listings we had found in related work. Table 2.10 presents the final grouping, indicating the group name and the grouped topics from Table 2.4. A description of each group is stated below.

- **Algorithm representations**: This group consists of topics related to other ways of constructing and representing algorithms, not necessarily using a programming language. Examples include pseudocode and flowcharts.

Table 2.10: Final grouping of the most covered topics in Brazilian public universities. Table sorted alphabetically by the group name.

Group	**Topics** (Table 2.4)
Algorithm representations	Algorithm representation forms
Basic concepts of algorithm construction	Variables, constants, and assignments Arithmetical, logical, and relational expressions Data input/output
Composite variables	One-dimensional homogeneous composite variables Multidimensional homogeneous composite variables Heterogeneous composite variables
Control structures	Conditional commands Repetition commands
Functions, scope, and parameter usage	Functions, modularization, subprograms Scope of variables and parameter usage
Recursion	Recursion

- **Basic concepts of algorithm construction**: This group consists of primordial topics necessary for the construction of simple algorithms. Examples include variables, constants, basic data types, and expressions. Assignment and input/output (from keyboard) commands are also present in this group.
- **Composite variables**: This group consists of composite types of data such as vectors, strings, matrices, and structures. Other types that depend on the programming language (e.g., lists, tuples, dictionaries) are also present in this group.
- **Control structures**: This group consists of commands for selection and iteration of code. These structures can vary based on the programming language or algorithm representation.
- **Function, scope, and parameter usage**: This group consists of concepts related to functions, subprograms, and modularization. Since scope of variables and parameter usage is closely related, we decided to also include them in this group.
- **Recursion**: This group consists of the specific topics regarding recursion. Again, we decided to create a separate group for this because the analyzed CS1 syllabi listed them separately. Recursion is also presented as a specific topic in related work.

2.6 Limitations and Threats to Validity

The main limitations regarding this work are consequences of the restrictions related to data collection explained in Section 2.3. Since our data collection and analysis had to be done manually, we had to limit our assessment. Choosing Brazilian public universities provides a broad context of assessment, since it covered all national geographical regions. Brazilian public universities are also the most present in international rankings. However,

there are other types of higher education institutions in the country, especially private universities and colleges. It is possible that our results could be different if we assessed data from these other institutions. Another limitation factor was the availability of the documents from the universities' websites. Data from public universities that we were not able to retrieve could also have changed our results.

We identify three main threats to the validity of this research. First, we used our empirical knowledge to identify and list the distinct covered topics present in each syllabus. Second, the dominance of the procedural paradigm taught in CS1 courses influenced in our discussion about the presence of topics in the syllabi. Third, our inferring methodology about the programming languages taught was vastly dependent on each syllabus. Since the analyzed syllabi were not homogeneous (sometimes not even listing the bibliography) and included documents ranging from as far as 2006, this could also have impacted on our results. We tried to mitigate these threats by comparing our topics with others retrieved from related work, and by applying a filter to analyze the programming languages from syllabi from 2018 to 2022.

2.7 Conclusions

In this work, we presented an assessment of characteristics of CS1 courses from Brazilian public universities. Our focus was the most covered topics, but we also reported the most common course names, when undergraduates take the course, time that is dedicated to teaching the course, and the programming paradigms and languages taught. To answer each research question, we gathered data directly from the public universities' websites, searching for undergraduate Computer Science programs listed in the Formation Guidelines for Computer Science Undergraduate Programs [149]. The main document analyzed was the syllabi of CS1 courses.

In total, our results derived from 225 syllabi within 95 Brazilian public universities. The most covered topics were from concepts related to the procedural programming paradigm. We identified 12 topics among the most frequent and grouped them in 6 categories: Algorithm representations; Basic concepts of algorithm construction; Composite variables; Control structures; Functions, scope, and parameter usage; and Recursion. We also concluded that the CS1 course is within the first scholarly year in 99% of the Brazilian public universities. The most common names of CS1 courses vary, and some of them have numbered entries that implies subsequent courses with the same name in the curriculum. Regarding the total class hours, we identified that it is dependent on the course being semiannual or annual, but in average, the latter is double the former. Lastly, we concluded that the most common taught programming paradigm is the procedural one, and the most common programming language is C.

2.8 Afterword

Regarding the thresholds outlined in Section 2.4, we initially selected 10% of the total 225 syllabi as a promising threshold to identify the most covered topics. However, the resulting

21 topics exhibited a wide range of occurrences, from 204 to 22 (Table 2.4). Given this variability, which we believed could obscure our objective of identifying the *most covered* topics, we opted to apply a stricter threshold of 33% of the total syllabi. These results mainly align with this thesis by enabling an external validation: the final grouping of the most covered topics (Table 2.10) encompasses almost all categories identified in MC³. This discussion is further explored with the assessment of SO 1 in Chapter 5.

The assertions that C is the most used language and that the procedural paradigm is the most prevalent should be considered within the limitations of this chapter's study. As outlined in Section 2.6, our research was solely based on publicly available documents. Additionally, the time frame should also be acknowledged: this thesis was defended in 2024, while the study was conducted in 2022. To gain a clearer understanding of whether, or why, these remain the most widely used, further research is necessary. A potential approach could involve directly surveying nationwide CS1 instructors or CS program coordinators regarding the programming languages they currently employ.

Chapter 3

When Test Cases Are Not Enough: Identification, Assessment, and Rationale of Misconceptions in Correct Code (MC³)[1]

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.

3.1 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

[1]This paper was published in the Brazilian Journal of Computers in Education (RBIE). DOI: `https://doi.org/10.5753/rbie.2023.3552`

their curricula. While these classes are predominantly found in programs within the Science, Technology, Engineering, and Mathematics (STEM) field [25, 40], they are also being introduced in other areas, including Law [123].

When considering the context of CS teaching, one of the common courses focuses on introductory programming concepts, commonly referred to as CS1 [8]. CS1 courses are often associated with challenges, including high failure and dropout rates [24, 76, 140]. 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 [65] 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 [88].

In the educational context, autograders play a crucial role in grading assignments, relieving instructors of some of their workload and conserving resources [46, 53, 65]. However, the use of autograders can also influence the development of undesirable habits in students. Baniassad et al. [10] 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 [68] or when students mistakenly assume that autograders cannot make mistakes, resulting in overconfidence [66].

One common application of autograders in CS1 courses is to verify if the output of a code matches the expected predetermined output [103]. This approach fosters research aimed at understanding and enhancing the teaching and learning of concepts based on code correctness [7, 16, 83, 94]. However, in CS1, even code that produces the desired outcome can exhibit undesirable characteristics that experienced programmers would typically avoid [39, 126]. Examples of such characteristics include higher code complexity resulting from the redundant use of syntactic constructs, such as conditional statements or nested loops [67, 114, 134]. 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 [39, 74]. 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 3.1. 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 instruc-

tions within its body to not alter the correct output. Code 3.2 provides an alternative implementation that resolves the misconception demonstrated in Code 3.1.

```
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 3.1: Example of unnecessary `else`

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

Code 3.2: Code 3.1 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 [106]. In CS1, research on misconceptions typically focuses on identifying and classifying errors made by students, including syntactic, semantic, or logical errors [6, 28, 30, 54]. 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 MC^3 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 3.2 presents the background and related work. The methodology used are described in Section 3.3, followed by the obtained results in Section 3.4. We discuss the results in Section 3.5. Section 3.6 details the limitations and threats to validity of this research. Lastly, we present the conclusions in Section 3.7.

3.2 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.

3.2.1 Background

There are various terms used to describe faulty comprehensions of concepts taught in CS1 classes. Qian and Lehman [106] 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) [3, 4]. A CI is an assessment tool specifically designed to identify and address misconceptions within a specific domain of knowledge [4, 28], often in the form of a multiple-choice questionnaire. Almstrum et al. [4] 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 [28], while there are ongoing research for Python [54] and Java [30]. Additionally, Tew and Guzdial [131] 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 [134] 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 [31]. However, there is a risk of students becoming overly reliant on automated feedback [10]. Baniassad et al. [10] 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. [93] 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. [82] 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[2], an autograder developed by the Federal University of Amazonas.

[2] `https://codebench.icomp.ufam.edu.br/`

3.2.2 Related Work

By analyzing students' answers to exams and interviewing CS1 instructors, Caceffo et al. [28] 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. [54] conducted an analysis to examine the applicability of the misconceptions identified by Caceffo et al. [28] 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. [6] expanded upon the results obtained by Gama et al. [54] 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. [7] 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 [83]. PyTA promotes static code analysis [142] 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. [39] 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 [134] 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. [101], 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)[3], developed by Keuning et al. [75], is an Intelligent Tutoring System [138] 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. [39]. 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. [92] 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 3.1: Comparison of this research with the presented related work.

Research	**Errors/Misconceptions analyzed**	**Analysis of correct code**	**Language**
Caceffo et al. [28]	15	-	C
Gama et al. [54]	28	-	Python
Araujo et al. [6]	21	-	Python
Araujo et al. [7]	Not mentioned*	-	Python
Liu and Petersen [83]	Not mentioned*	-	Python
De Ruvo et al. [39]	16	✓	Java
Ureel II and Wallace [134]	200	✓	Java
Porfirio et al. [101]	37	✓	C
Keuning et al. [75]	19	✓	Java
Oliveira et al. [92]	25	✓	Java
This research	45	✓	Python

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

Table 3.1 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 [6, 7, 28, 54, 83]. 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 [39, 75, 92, 134].

[3] `http://hkeuning.nl/rpt/`

3.3 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 3.1 summarizes the methods used in this work.

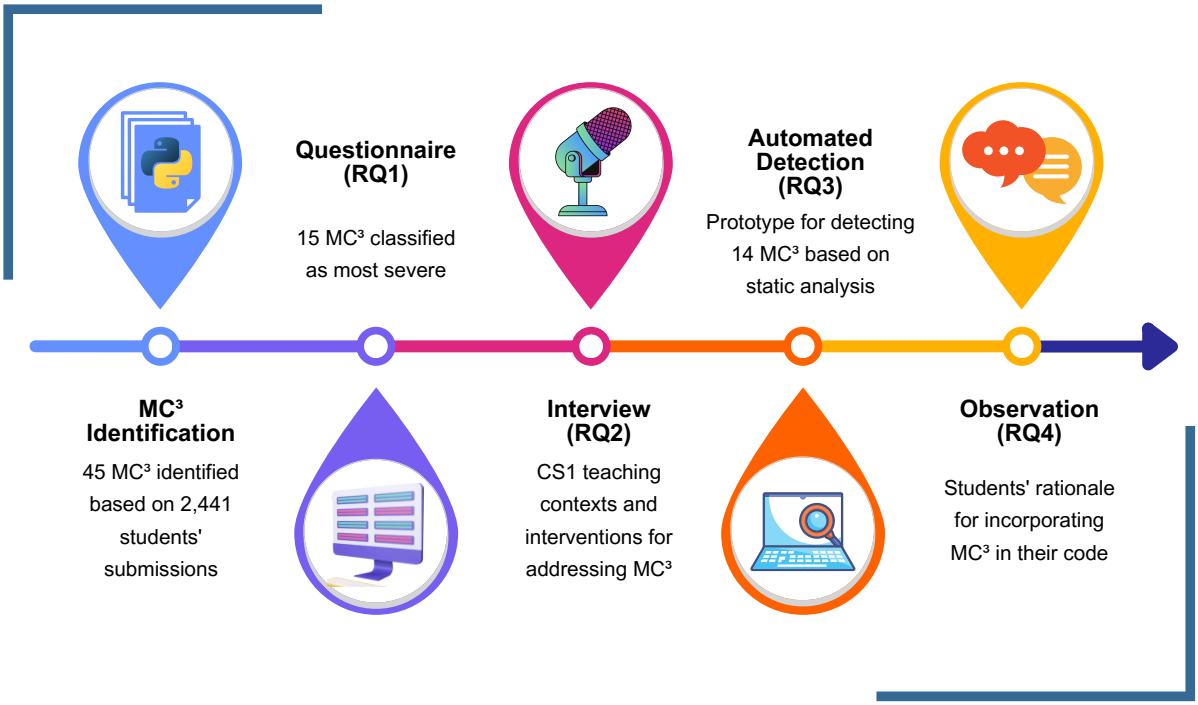

Figure 3.1: Description of the methods used in this research.

3.3.1 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[4], 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 MC^3 in code deemed correct by an autograder, we collected and analyzed the students' submissions to MC102 assignments.

3.3.1.1 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 3.4.

3.3.1.2 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

[4] `http://ic.unicamp.br/~susy/`

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.

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. [54]. We analyzed a total of 2,441 submissions, resulting in an initial list of 45 MC³ divided into 8 categories, detailed in Section 3.4.

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 [81] 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[5].

3.3.2 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.

3.3.2.1 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.

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

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 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.

3.3.2.2 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 [81]. 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.

3.3.3 RQ2: Addressing MC³ in CS1

We conducted semi-structured interviews [81] 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.

3.3.3.1 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 MC^3 and how the instructor handles them.

4. A set of questions related to teaching and learning interventions aimed at mitigating the occurrence of MC^3 and the instructor's perspectives on their implementation in the classroom.

3.3.3.2 Data Analysis

After the interviews were concluded, each answer was compiled and analyzed individually using content analysis [81]. 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 MC^3 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 MC^3 in CS1 classes, such as the use of autograders capable of detecting MC^3 and the implementation of Active Learning [22] techniques specifically targeting these coding behaviors.

3.3.4 RQ3: Frequency Distribution of MC^3

Building upon the insights obtained from RQ1 and RQ2, our next endeavor was to explore the automatic detection of the most severe MC^3. Automating this procedure would not only facilitate its integration with an autograder but also provide the opportunity to assess the frequency of MC^3 in our dataset, as well as in other datasets. Additionally, analyzing the distribution of MC^3 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.

3.3.4.1 Data Collection

To accomplish the automatic detection, we leveraged static analysis techniques [142]. Specifically, we employed Python module AST[6] for this purpose. This module provides tools to inspect and modify the Abstract Syntax Tree [77] of Python code, which is generated after parsing the syntax but before compiling the bytecode.

Out of the 45 identified MC^3, a subset of 15 was deemed the most severe (refer to Section 3.4 for detailed information). We chose to prioritize the implementation of the automatic detection process for this subset. The implementation was carried out exclusively

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

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.

3.3.4.2 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 MC^3, counting whether the MC^3 was present or not in the code. Subsequent occurrences of the same MC^3 in the same code were not counted twice.

During the implementation process, we encountered certain challenges in the automated detection of certain MC^3. Specifically, decisions regarding code that was deemed redundant, non-significant, or unnecessary still required instructor intervention. In such cases, we employed threshold values to determine if the code exhibited the associated MC^3 or not. Further details on the grouping of CS1 concepts and the thresholds used can be found in Section 3.4.

3.3.5 RQ4: Why Students Code with MC^3

During the identification of the MC^3, 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 MC^3 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 MC^3 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[7].

3.3.5.1 Data Collection

A semi-structured observation [37] methodology was employed to gather the data. Cohen et al. [37] 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 MC^3 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 MC^3 in the code was examined after the submission deadline. In this research, all code submissions were checked for the presence of any MC^3, regardless of the severity of the behaviors. Following each assignment analysis, one researcher conducted brief conversations of approximately

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

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.

3.3.5.2 Data Analysis

The field notes that were developed during the observation of the instructor and the conversations with students were analyzed using content analysis [81]. 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.

3.4 Results

This section presents the results obtained, following the same order as described in Section 3.3. 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 [116].

3.4.1 MC³ Identification

As mentioned in Section 3.3, 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 3.2 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.

Table 3.2: 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**

After analyzing all 2,441 submissions, we identified a total of 45 MC³, which were split into eight distinct categories. Table 3.3 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 3.2 correspond to the assignments appointed to students. Given the 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.

3.4.2 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 3.3 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 3.3) 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[8].

Table 3.3: 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

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.

[8]**Thesis note**: in this chapter, we exemplify and explore only the 15 most severe MC³. Chapter 5 presents Table 5.1, which lists all the teaching interventions developed in this thesis. Additionally, Appendix A provides detailed information for all 45 identified MC³, including examples, rationale, consequences, and possible classroom interventions.

3.4.2.1 Set 1

This example, denoted in Code 3.3, contains 5 MC[3]: 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.

```
1  a = int(input())
2  def foo(a):
3      a = a / 3.5
4      round(a, 2)
5      return a
6
7  b = int(input())
8  c = int(input())
9  def max():
10     if b >= c:
11         return b
12     return c
13
14 x = foo(a)
15 y = max()
16 print(x, y)
```

Code 3.3: Examples of MC[3]: A4, D4, G4, G5, and H1

3.4.2.2 Set 2

This example, denoted in Code 3.4, contains 4 MC[3]: 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.

```
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 3.4: Examples of MC³: B6, B8, B9, and B12

3.4.2.3 Set 3

This example, denoted in Code 3.5, 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.

3.4.2.4 Set 4

This example, denoted in Code 3.6, 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.

```
numList = []
for i in range(9999):
    a = int(input())
    if a == 0:
        break
    numList.append(a)

numMax = max(numList)
for j in range(1):
    print(numMax)

for k in range(numMax):
    print(k + 1)
    k += 2

while numMax != 0:
    print(numMax)
    numMax = numMax - 1
    if numMax == 0:
        break
```

Code 3.5: Examples of MC³: C1, C2, C4, and C8

```
numList = []
num = int(input())
while num != 0:
    numList.append(num)
    num = int(input())

totalSum = 0
for item in numList:
    totalSum += item

if numList == [1, 1, 1]:
    print(3)
elif numList == [2, 2, 2]:
    print(6)
elif numList == [1, 2, 3, 4, 5]:
    print(15)
```

Code 3.6: Examples of MC³: E2 and F2

3.4.3 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 MC³, and means to verify if students are implementing the feedback related to MC³ in their code (such as questionnaires).

As an additional intervention method for addressing MC³, eight instructors expressed interest in using Peer Instruction (PI) [38]. PI is an Active Learning technique that promotes meaningful discussions among students regarding different comprehension aspects of a topic. In the context of MC³, 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 MC³, 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.

3.4.4 MC³ Frequency Distribution

As mentioned in Section 3.3, 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 MC³ frequency in Table 3.4.

By using the Python AST module, we managed to implement an automatic detection for 14 out of the 15 MC³ 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 MC³. For instance, regarding the MC³ 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 MC³ 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 MC³ 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 3.4 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 3.3. 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 3.5, we delve into the reasons behind these variations and discuss how they impact our findings.

Table 3.4: Frequency distribution of the most severe MC[3] 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 MC[3]. Table is sorted decreasingly by Total.

MC[8]	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

3.4.5 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 3.4).

3.4.5.1 Conversations with Students

Based on the researcher's notes, we identified students' reasons for 10 MC[3]: A4, B8, B9, C1, C8, D4, E2, G4, G5, and H1. In general, two recurring comments were observed regarding these MC[3]. Firstly, students expressed a concern about ensuring that their code passed all test cases. This led to the development of MC[3] 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. MC[3] 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.

3.4.5.2 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.

3.5 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 MC^3 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 2022[9] to calculate the frequency distribution of MC^3. 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.

3.5.1 MC^3 Severity and Reasons for Occurrence (RQ1 and RQ4)

The identification and assessment of the MC^3 followed some of the guidelines described by Almstrum et al. [4]. 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 MC^3. In the following subsections, we discuss the most severe MC^3 within each category (highlighted in Table 3.3), incorporating the opinions collected from CS1 instructors and explaining why MC102 students incorporated those MC^3 into their code.

3.5.1.1 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 MC^3 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

[9] **Thesis note**: In the original article, this year was incorrectly cited as "2023." I have taken the liberty of correcting it to the accurate year in this thesis.

data from external sources. While we also identified this category present in other work [39, 75, 92], 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.

3.5.1.2 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* [134]. Soloway and Ehrlich [126] 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 [75] has rules to extract duplicate declarations inside `if/else` statements, and Oliveira et al. [92] 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 3.4 illustrates that B9 did indeed appear throughout the analyzed assignments. De Ruvo et al. [39] 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.

3.5.1.3 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 [127].

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` [75]. Oliveira et al. [92] 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.

3.5.1.4 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.

3.5.1.5 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 [51, 126] 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.

3.5.1.6 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.

3.5.1.7 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.

3.5.1.8 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. [39] 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.

3.5.2 Addressing MC³ in CS1 Classes (RQ2 and RQ3)

The decision to study MC³ in Python was well-founded as this language is widely utilized [13, 58]. 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.

3.5.2.1 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 [30]. However, despite these challenges, research has demonstrated positive outcomes when employing Active Learning techniques [120, 121]. For example, students achieved better results compared to traditional teaching methods [122], and failure rates were reduced [102]. 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.

3.5.2.2 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.

3.5.2.3 Insights from MC³ Automated Detection

The frequency distribution of MC³, as presented in Table 3.4, 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 [39], could also detect MC³, particularly those in category **A**. However, these tools often provide extensive feedback that may overwhelm CS1 students [74, 75]. 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 [13, 118]. Evidence suggests further research to explore and develop viable and comprehensive approaches to integrating automated MC³ detection in CS1 courses.

3.6 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 3.2) and the identified MC³ categories (Table 3.3) represented topics listed as most covered in typical CS1 courses [13, 118], 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.

3.7 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 MC^3 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 MC^3. Additionally, nine of these instructors took part in a semi-structured interview, aimed at exploring different strategies to address MC^3 in various teaching and learning contexts within CS1 courses. Furthermore, to gain insights into the reasons why students incorporated MC^3 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 MC^3 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 MC^3 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 MC^3. 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 MC^3 in CS1 classes: integrating the detection of these misconceptions into an autograder to provide formative feedback; incorporating MC^3 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 MC^3 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 MC^3 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 [11, 51], researchers argue that stimulating behaviors related to the latter is more beneficial for CS1 students [39, 70, 74]. Therefore, we advocate that addressing MC^3 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 MC^3 occurrence serve as basis for creating feedback messages that can improve teaching and learning outcomes [31]. 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 [45].

3.8 Afterword

The names assigned to each of the eight MC³ categories, as described in Section 3.4, were inspired by the taxonomy used in the work of Gama et al. [54]. Adopting a taxonomy from an established study on Python misconceptions appeared to be a suitable approach. However, we had to develop new categories to more accurately describe our identified groups of MC³ for those we could not find a proper group in the related work. As earlier mentioned in Chapter 2, the MC³ categories match the grouping of the identified most covered topics, and this match will be further discussed in Chapter 5.

Section 3.4 outlined the CS1-related topics covered in each of the six assignments used as the basis for manually identifying the 45 MC³ (Table 3.2). However, as noted, the submission deadline for an assignment is typically three weeks, allowing students to learn new concepts while still working on their previous assignments. Additionally, due to the COVID-19 pandemic, these deadlines were exceptionally extended for that academic term of MC102. Considering the deadline for the last assignment we used in our analysis, we expect that students could have already been exposed to the following topics: basic concepts of computer organization; data I/O; arithmetic, logical, and relational expressions; conditional commands; repetition commands; lists, tuples, dictionaries, strings, and matrices; and functions and variable scope. This factor provides further explanation on why we identified MC³ related to topics beyond those listed in Table 3.2.

Chapter 4

From forest to leaves: assessing and addressing misconceptions in programming novices' correct code[1]

Abstract: Use of automatic grading systems help alleviate instructor's workload in introductory programming courses (CS1). However, novices may over rely on correctness and still exhibit coding behaviors that potentially indicates misunderstandings of the concepts taught in the course. In this work, we investigate students' code for a Python CS1 course to assess occurrence of misconceptions present in code that already produces the expected results. These were named Misconceptions in Correct Code (MC^3). We conduct a large-scale study analyzing over 40,000 correct submissions from students in eight academic terms of a CS1 course, assessing characteristics of the frequency distribution of MC^3. Our results indicate that, by the end of the CS1 course, a fraction of novices still struggles to grasp fundamentals regarding conditional and repetition commands, as well as exhibits a lack of effort to produce code that promotes readability and maintainability. Evidence also suggests that MC^3 are persistent across the academic term, indicating that students do not correct these behaviors by themselves. We propose that CS1 students should receive feedback on their correct code especially when they might not fully understand the course's learning objectives. In order to do that, instructors should also implement formative artifacts to engage students in understanding the consequences that may befall lest those misconceptions are not addressed.

4.1 Introduction

Over the years, numerous studies have focused on identifying factors that hinder the learning process for novice programmers. These studies often focus on introductory programming courses, commonly referred to as CS1 and CS2 [8]. Typically offered during the early stages of undergraduate programs, these courses play a crucial role in laying the foundation for students' programming skills. Despite ongoing debates on the specific

[1]This paper was not yet published. However, it was constructed as a journal article for future submission.

topics that should be covered in these courses [13, 62, 118], they consistently face common challenges, including high dropout and failure rates [2, 76, 98].

A particular area of research aims to identify, understand, and address common mistakes made by novice programmers. In the context of CS1, these mistakes are described using various terms such as faulty comprehensions, issues, errors, and misconceptions [106]. The differences in these terms reflect the specific aspects of code that researchers choose to focus on. Some studies investigated how students learn to program, targeting specific levels of code complexity [41, 42, 126, 128]. Others concentrated on compiler error messages, seeking to improve the presentation of these messages to novice programmers [7, 15, 99]. Similarly, other research identified students' misconceptions through analysis of incorrect exam or assignment solutions [6, 28, 54]. However, some studies focused on identifying misconceptions in code that already produced the expected results [39, 75, 92, 114, 119]. These studies highlighted the importance of addressing these misconceptions even in seemingly correct code, as they possibly indicate incomplete or incorrect understanding of the underlying concepts. Independent of the focus, all research in this category is important because not only do these misconceptions hinder students in CS1, but they can also continue to manifest in later courses [72].

The initial step in understanding common mistakes among novice programmers often involves observation processes. Large-scale studies have been pivotal in identifying the frequency distribution of error messages [34, 104], analyzing demographic contexts regarding error messages [141], and studying the time it takes students to rectify errors [5]. Furthermore, misconceptions can be addressed through concept inventories, validated assessment instruments designed to pinpoint and rectify misconceptions within specific knowledge domains [4]. These inventories may focus on particular topics such as recursion [61], general concepts of CS1 regardless of programming languages [71, 131], or focus on specific languages such as C [28]. Ongoing research is exploring concept inventories for languages like Java [30] and Python [54]. Once sets of misconceptions are identified and validated, they can be incorporated into teaching in other ways. These include providing formative feedback generated through automated detection of misconceptions [47, 50, 59], addressing misconceptions through educational media [20, 97, 100], creating repositories that present fine-grained information on misconceptions [36], and developing sets of questions that educators can integrate into their classrooms [55].

In a previous study [119], we explored a specific type of misconception concerning CS1 topics in Python, referred to as Misconceptions in Correct Code (MC^3). MC^3 are coding behaviors that indicate an incomplete or incorrect understanding of CS1 topics. These coding behaviors are present in student code that already produces the desired results. We also identified that MC^3 are identified in teaching environments that use autograders [65], which are automated assessment tools that often grade students' submissions mainly for code correctness [103]. For example, students may mistakenly include redundant logic in an `else` clause, believing it to be mandatory. In this case, students exhibit a faulty comprehension of conditional commands albeit their code produces the expected output. In this work, we conducted a large-scale study using automated MC^3 detection in a CS1 course at a Brazilian state university to assess the persistence of MC^3 and further analyze the frequency behavior of their occurrences.

This study aimed to address the following research questions:

- **RQ1**: *How are the occurrences of MC³ distributed throughout academic terms of a CS1 course?*

- **RQ2**: *In what ways does the passage of time affect MC³ occurrences throughout the academic term?*

Our large-scale analysis, encompassing over 40,000 student code submissions across eight academic semesters, reveals that while MC³ vary in frequency, students do not appear to naturally correct these behaviors over the academic term of the CS1 course. Further analysis shows that the frequency distribution of MC³ remains persistent throughout the academic term. In other words, students begin to develop MC³ at some point during the course, and these misconceptions may even carry over into later CS courses. Additionally, MC³ occurrences do not seem to be influenced by the proximity to assignment deadlines, nor are they dependent on previous occurrences.

We believe that the findings of this study can contribute to enriching the community discussion on the general assessment of student learning in CS1. MC³ present challenges as they manifest in an already functioning code, influencing students' perceptions of the importance of learning about these behaviors.

The remainder of this paper is structured as follows: Section 4.2 provides background information and presents related work. Section 4.3 presents more information about MC³. The methods are described in Section 4.4, followed by the obtained results and its discussion in Section 4.5. Section 4.6 described the implications of the results for CS1 teaching. Section 4.7 describes the limitations and threats to validity, and the conclusions are presented in Section 4.8.

4.2 Background and related work

This section provides a review of previous research addressing challenges similar to those examined in this study. We begin by discussing studies that investigated why novices elaborate code that is correct but also presents poorly constructed structures. Following this, we explore large-scale studies that focused on analyzing students' errors. At the end of this section, a comparison of how this work differs from related ones is presented.

4.2.1 Assessing correct but poorly constructed novices' code

Soloway and Ehrlich [126] proposed that novices and expert programmers differ in their knowledge of programming plans and rules of programming discourse. Programming plans involve action sequences, such as item search loops, while programming discourse rules encompass general conventions expected by experienced programmers, like naming variables or functions appropriately. According to the authors, a program can be technically correct but difficult to read or write if the programmer deviates from these discourse rules. In their empirical study involving 180 programmers, including novices and experts,

Soloway and Ehrlich compared the performance of evaluating programs constructed with and without adherence to these rules. They found that expert programmers performed similarly to novices when prompted with programs that violated discourse rules.

Joni and Soloway [70] proposed an alternative approach to assessing students' code by prioritizing readability over efficiency, in which the latter is generally used to evaluate a working but poorly constructed program. They argued that readable code follows a concise plan structure, minimizing confusion for other programmers. In their study, they analyzed 57 examples of working but poorly constructed code from around 200 students in an introductory Pascal course. Applying the concepts of programming plans and rules of programming discourse, they found that over 90% of the analyzed code violated these rules, leading to reduced readability. The authors identified specific rules related to variable management, `if` statements, `while` loops, and students' rationale for code construction.

Fisler et al. [51] advocated for the teaching of problem decomposition and programming plans in CS1 courses across undergraduate programs. Their multinational study introduced modernized plan-composition problems in CS1 courses, encompassing both imperative and functional programming paradigms. Students were tasked not only to elaborate solutions to these problems but also rank their preference on preestablished solutions. The authors highlighted the impact of library functions on students' plan composition, as well as how programming paradigms influence the creation of intermediate variables.

In a related study, Izu et al. [69] aimed to improve program comprehension skills in novice programmers through the principles of program composition and plans. This program comprehension involves code reading, interpretation, and explanation skills. Drawing from a review of the literature and insights from experienced instructors, the Working Group developed a map of learning activities and possible learning trajectories. These resources were designed to help instructors decompose complex tasks and identify the targeted concepts for program comprehension.

4.2.2 Large-scale studies in CS1

Altadmri and Brown [5] analyzed Java code from more than 250,000 students worldwide, comprising several different institutions within an academic year. These codes were obtained from the Blackbox dataset [26], a large repository that collects data from the BlueJ IDE, mainly used by novice programmers. The authors assessed 18 types of mistakes, including syntax (in which the program does not compile) and semantic (in which the program does compile but produces a wrong output) errors. In their findings, Altadmri and Brown noted that, in addition to mismatching brackets or quotations, semantic errors were among the most frequent. The frequency of these errors varied throughout the academic term, with syntax errors being more frequent at the beginning and then diminishing, whereas semantic errors exhibited the opposite trend. In addition, students tended to spend less time rectifying syntax and type errors compared to semantic errors. This observation suggests that semantic errors pose greater challenges for students to resolve, as indicated by the authors' analysis.

Pritchard [104] conducted a study investigating syntax and run-time errors in both Python and Java. The datasets analyzed comprised 640,000 submissions with errors in Python and approximately 4 million in Java. These datasets were sourced from CS Circles [105] and Blackbox. While the authors identified the most frequent error messages in each language, their primary objective was to determine if the error frequency adhered to a Zipf-Mandelbrot distribution. This distribution implies a certain inherent order to error messages, ranging from most to least frequent. In their findings, Pritchard demonstrated empirically that the error frequencies indeed resemble the Zipf-Mandelbrot distribution, shedding light on the underlying structure of error occurrence in programming languages.

In a similar study, Caton et al. [34] analyzed 32,000 Java submissions from novice programmers using the CodeRunner platform. The authors aimed to explore the relationship between syntax and compiler errors and cases where code runs but produces incorrect results. By categorizing error frequencies into different states, Caton et al. employed a combination of Markov chain analysis and Association Rule Mining to uncover patterns and relationships. They observed that simple errors that led to a faulty compilation often recurred successively. Furthermore, these simple errors were found to increase the likelihood of code that compiled, but produced the wrong results by 50%. The authors advocated that teaching contexts that use autograders should present materials to make students aware of these error scenarios.

Wang et al. [141] conducted a large-scale randomized controlled trial to evaluate the effectiveness of error messages in introductory programming courses. Their objective was to compare traditional Python error messages with two novel approaches that utilized OpenAI's GPT to generate enhanced error messages. The study involved over 8,000 students from 146 countries, resulting in the generation of over 400,000 errors across different types of messages. In assessing effectiveness, Wang et al. analyzed the time students took to fix these errors. Additionally, the authors analyzed effectiveness by demographics, including different human development indices, gender, and previous programming experience. Overall, the findings suggested that error messages generated by GPT were consistently more helpful to students across diverse demographics compared to traditional Python error messages.

4.2.3 Comparison with our work

Table 4.1 provides a summary of the contributions of the most relevant previous studies compared to ours. The table outlines the corpus analyzed to make up each study, categorized by the types of mistakes examined. Our study differs from those that conducted large-scale analyzes [5, 34, 104, 141] in that we focused on code that already produces the expected results, thus excluding syntax or semantic errors. By expanding the analyzes conducted in our previous work [119], this research contributes to both the areas of large-scale investigations and to the studies of misconceptions in CS1.

Table 4.1: Comparison of this research with the described related work.

Research	Corpus		Mistakes	Language(s)
	Type	Analyzed	Type(s)	
Altadmri and Brown [5]	Students	250,000	Syntax, semantic	Java
Pritchard [104]	Code	Over 4 million	Syntax, compiler	Java, Python
Caton et al. [34]	Code	32,000	Syntax, compiler, semantic	Java
Wang et al. [141]	Students	8,000	Syntax, compiler	Python
Silva et al. [119]	Code	2,441	MC^3	Python
This research	Code	40,882	MC^3	Python

4.3 Misconceptions in Correct Code (MC^3)

In this section, we begin by presenting the details of the CS1 course that served as the basis for our studies. We then briefly summarize the results of our previous work [119], describing the process used to identify the list of MC^3 employed in this research.

4.3.1 The Algorithms and Computer Programming course

For both studies, we chose a CS1 course called Algorithms and Computer Programming (MC102) from a Brazilian state university as basis for research. MC102 typically enrolls approximately 600 students per academic term. The course has been using Python 3 since 2018, with instruction focused on the imperative paradigm. MC102 classes are generally coordinated between professors, where all students follow the same syllabus and complete identical exams and practical assignments. Bachelor's programs in this environment are mostly in engineering disciplines. MC102 follows a six-hour per week structure, with four hours allocated to theory lectures and two hours to practical sessions. Instructors are offered a deck of lecture slides and follow a predetermined calendar, ensuring that all different classes advance at the same pace. Practical sessions are facilitated by teaching assistants and serve primarily to provide guidance and assistance with the completion of practical assignments.

The syllabus for MC102 can be described in the following order: 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.

4.3.2 Identification and initial assessments of MC^3

In CS1, research on misconceptions typically centers on identifying and classifying the errors students make, which may include syntactic, semantic, or logical errors [28, 30, 106, 128]. However, these studies are not necessarily limited to correct code. Thus, we chose to narrow our focus and specifically examine faulty or incomplete understandings manifested in code that already produces the expected results. As a result, we established a subgroup termed Misconceptions in Correct Code (MC^3). In other words, while MC^3 represents

a subset of misconceptions within the broader CS1 research field, not all misconceptions studied in this field can be classified as MC³.

The identification process involved a manual analysis of 2,441 student submissions. By grouping similar coding behaviors, we identified a total of 45 MC³, which were then organized into eight distinct categories. Each MC³ was assigned an identifier, consisting of the category letter and a sequential number. The categories are as follows: A) Variables, identifiers, and scope (MC³ A1 to A8); B) Boolean expressions (MC³ B1 to B12); C) Iteration (MC³ C1 to C8); D) Function parameter use and scope (MC³ D1 to D4); E) Reasoning (MC³ E1 and E2); F) Test cases (MC³ F1 and F2); G) Code organization (MC³ G1 to G6); and H) Other (MC³ H1 to H3).

After identifying the initial list of MC³, we conducted assessments with both CS1 instructors and students to explore the underlying causes of these misconceptions. Additionally, we sought to determine which MC³ were most critical to address, prioritizing those that posed greater severity and required correction during the CS1 course. These assessments were carried out through questionnaires, participant observations [37], and semi-structured interviews [81]. The design of these processes were based upon the guidelines for identifying misconceptions established by Almstrum et al. [4]. Specifically, we analyzed open-ended assignments from MC102, consulted experts in the field (CS1 instructors), and conducted observations with CS1 students within the context where these misconceptions arose.

4.3.2.1 Most severe MC³

Following the initial assessments, 15 of the original 45 MC³ were identified as the most severe. Table 4.2 presents their corresponding IDs and names. It is important to note that, for the sake of consistency with the original list (which is omitted in this study), the names and IDs of these 15 MC³ remained unchanged.

Table 4.2: List of most severe MC³ identified in our previous work [119]. Table is sorted in lexicographical order by MC³ ID.

ID	Name
A4	Redefinition of built-in
B6	Boolean comparison attempted with `while` loop
B8	Non utilization of `elif/else` statement
B9	`elif/else` retesting already checked conditions
B12	Consecutive equal `if` statements with distinct operations in their blocks
C1	`while` condition tested again inside its block
C2	Redundant or unnecessary loop
C4	Arbitrary number of `for` loop execution instead of `while`
C8	`for` loop having its iteration variable overwritten
D4	Function accessing variables from outer scope
E2	Redundant or unnecessary use of lists
F2	Specific verification for instances of open test cases
G4	Functions/variables with non significant name
G5	Arbitrary organization of declarations
H1	Statement with no effect

Code 4.1 exemplifies six MC[3]: A4, B6, B8, B9, B12, and G4. The variables **num1** and **num2** were likely declared with non-significant names (G4). In line 4, the variable **sum** redefines a Python built-in function (A4). Additionally, the **while** condition in line 5 executes only once, indicating a potential confusion with an **if** statement (MC[3] B6). Furthermore, the **if-elif** sequence in lines 9 and 11 lacks a final **else** clause (MC[3] B8). In this case, the **elif** in line 11 could be simplified as an **else**. The **elif** in line 16 redundantly checks the opposite condition already addressed in line 14 (MC[3] B9), possibly indicating an incomplete understanding of **elif** behavior. Lastly, lines 20 and 22 verify the same **if** condition but perform different operations within their blocks (MC[3] B12), which could have been consolidated into a single **if** statement.

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

Code 4.1: Examples of MC[3]: A4, B6, B8, B9, B12, and G4.

Code 4.2 illustrates five MC[3]: C1, C2, C4, C8, and H1. In line 2, a **for** loop is declared, but it attempts to simulate a **while** loop by using an arbitrary number of iterations, indicating a misunderstanding between the two constructs (C4). Additionally, the **for** loop in line 9 executes only once, which suggests a misinterpretation of a loop's purpose (C2). The **for** loop in line 12 overwrites its iteration variable, **k**, within the loop body (C8), potentially reflecting the student's lack of awareness about how iteration variables are updated. In line 19, an **if** statement checks the **while** condition at the end of the loop's body (c1), which is redundant, as this test is inherently performed at the end of each **while** iteration. Lastly, line 25 contains a standalone **True** statement, which serves no functional purpose (MC[3] H1). Its occurrence may stem from other misconceptions such as the belief that the **else** clause is mandatory.

```
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
21
22 if numMax % 2 == 0:
23     print(numMax, "is␣even")
24 else:
25     True
```

Code 4.2: Examples of MC³: C1, C2, C4, C8, and H1.

Code 4.3 illustrates the remaining four most severe MC³: D4, E2, F2, and G5. The use of `numList` can be considered a redundant use of lists (E2), as it is only used to sum its values in lines 8 and 9. In this scenario, `totalSum` could have been calculated while reading the input. Additionally, the code demonstrates arbitrary organization of declarations (G5), with function definitions interspersed with variable manipulations. Ideally, functions should be defined at the beginning of the code for better structure. Furthermore, `CheckSum()` accesses the value of `numList`, which lies outside its scope (MC³ D4), a practice that should be avoided. Now, consider that the set of input from open test cases for the code was $I = \{\{1, 1, 1\}, \{1, 2, 3\}, \{1, 2, 3, 4, 5\}\}$ and the expected output was $O = \{\{3\}, \{6\}, \{15\}\}$. To achieve the correct result, the student ignored the value of `totalSum`, instead directly printed the expected values for each input (F2) in lines 12, 14, and 16. This suggests a potential misunderstanding of how the autograder system operates.

After conducting interviews with both CS1 instructors and students, two primary causes for the occurrence of MC³ were identified: incomplete comprehension of underlying concepts and a careless approach to coding. Many students expressed difficulties in understanding decision-making and iteration constructs, leading to MC³ in categories B and C. In contrast, other MC³ were frequently attributed to students focusing solely on passing test cases, neglecting proper code organization (MC³ D4 and G5) or the naming of variables and functions (MC³ A4 and G4). Additionally, it was also noted that the autograder's functionality contributed to the occurrence of MC³, particularly in cases such

as C1 and G5. Students reported concerns that the system might reject their solutions unless they included redundant checks, or that they deliberately disorganized their code to avoid plagiarism detection.

```
numList = []
num = int(input())
while num != 0:
    numList.append(num)
    num = int(input())

totalSum = 0
for item in numList:
    totalSum += item

def CheckSum():
    if numList == [1, 1, 1]:
        print(3)
    elif numList == [1, 2, 3]:
        print(6)
    elif numList == [1, 2, 3, 4, 5]:
        print(15)
CheckSum()
```

Code 4.3: Examples of MC³: D4, E2, F2, and G5.

4.3.2.2 Automated detection of MC³

Another contribution from previous work [119] was the development of a tool[2] for the automated detection of MC³ using static code analysis [142]. The tool is capable of detecting, with varying degrees of abstraction, 14 of the MC³ classified as most severe (with the exception of F2). It is important to note that some MC³ detection require specific configuration, which must be set by the instructor based on the criteria of the assignments.

- **C4**: requires the user to set the threshold to consider how many `for` loop iterations (declared inside the `range()` function) are needed to be present to flag the MC³.

- **E2**: requires the user to set the threshold to consider how many user-declared lists are needed to be present to flag the MC³.

- **G4**: requires the user to set a minimum character length for variables and function names. The MC³ is flagged if a percentage of total variables or functions that does not match the length criteria is met. This percentage is also defined by the instructor.

[2] `https://github.com/eryckpedro/mc4`

4.4 Methods

In this section, we outline the methods used in this research. Since the same data collection process was applied to both research questions, we begin by detailing this process. We then describe the investigation into the frequency distribution of MC³ across the academic terms of a CS1 course (RQ1). Following this, we present how the analyzes of MC³ persistence and the patterns in its occurrence were conducted (RQ2).

4.4.1 Data collection

We collected students' assignment submissions through the institutional autograder, which conducts dynamic analysis to evaluate whether each code passes pre-established test cases. In total, submissions from eight different academic terms were analyzed. To ensure that we focused solely on correct code, which is part of the definition of MC³, we removed submissions that did not pass all test cases. Following this filtering process, the submissions were classified by academic term, class, assignment number, and individual student ID. The frequency distribution of each MC³ was calculated using the developed tool described in Section 4.3.

We examined the presence of MC³ only in students' solutions to the practical assignments of MC102. It should be noted that the course grade formula has changed since 2018. In older terms, the total grade was composed of both practical assignments and partial exams. This changed in 2020, when the total grade began to consist only of the weighted average of practical assignments. Assignments are also changed for each term, sometimes with a variation in their total number. Consequently, certain topics in the syllabus were occasionally grouped together in the same assignment (e.g., strings and dictionaries), while others appeared across multiple assignments (e.g., matrices, recursion).

4.4.2 RQ1: MC³ frequency distribution

The primary objective of this phase was to explore how the occurrence of MC³ evolves throughout the academic terms of a CS1 course. This inquiry was motivated by the hypothesis that MC³ may initially manifest during the course but tend to diminish, or vanish entirely, as students gain programming experience. To evaluate this, we analyzed the frequency distribution of MC³ across multiple assignments from multiple academic terms of MC102.

To conduct the data analysis, the frequency distribution of each MC³ was calculated as a proportion of the total submissions for each assignment (already considering our filtering process). Then these results were grouped by sequential assignment number. This approach facilitated not only the evaluation of the proportion of submissions that incorporated MC³, but also the identification of how these occurrences are distributed alongside the academic term. Even though not all analyzed academic terms had the same number of assignments, all assignments followed the sequential order of the course syllabus (see Section 4.3). Consequently, we decided to group together assignments with equal number across all terms before calculating the frequency distribution of each MC³.

4.4.3 RQ2: Influence of the passage of time in MC³ occurrence

This phase sought to evaluate whether the occurrence of MC³, as identified in RQ1, persists throughout the academic term. Assessing persistence allowed us to deepen our understanding of the frequency distribution and identify potential avenues for future educational interventions. Additionally, we examined if contextual factors related to the practical assignments influenced the development of MC³, with particular attention to the proximity of submission deadlines. We hypothesized that students facing time constraints closer to the submission deadline would be more likely to develop MC³.

The analysis in this phase differed slightly from that of RQ1. To focus on the overall occurrence of MC³, we chose to aggregate all MC³ occurrences rather than analyzing them individually. However, this approach encountered two challenges: the first MC102 assignment and the MC³ related to functions/variables with non-significant names (classified as G4). The initial assignment in MC102 is solution-guided, intended to familiarize students with the autograder using a straightforward arithmetic exercise, resulting in minimal original code creation by students. The issue with MC³ G4 is that our automated detection tool requires the instructor's expertise to determine if the MC³ is present. Given the size of our dataset, manually assessing G4 occurrence in each code would be impractical. Consequently, both the first assignment and MC³ G4 were excluded from the analyzes conducted in RQ2. Additionally, we also opted to remove assignments 14 and 15 due to the their low number of submissions (Section 4.5 presents further details).

For the evaluation of MC³ occurrence in relation to the proximity to the assignment deadline, we first determined the number of days remaining until the deadline for all submissions containing at least one MC³. Subsequently, we calculated the total number of submissions in each category based on the number of days remaining until the deadline. This process also aggregated all assignments based on their number but was computed separately for each analyzed academic term.

4.5 Results and discussion

This section presents and discusses the results obtained in this research, following the structure outlined in Section 4.4. First, details are provided about the MC102 terms and classes whose submissions were analyzed. Then, we examine the frequency distribution of MC³ across academic terms (RQ1). Finally, we present the persistence analysis of MC³, exploring its occurrence behaviors and its relation to assignment deadlines (RQ2).

4.5.1 Terms and classes

Table 4.3 further presents contextual information on the eight academic terms analyzed in MC102. As mentioned above, the number of assignments varied in recent terms, despite maintaining the same syllabus. Each term is identified in a year-semester format (e.g. "2018-1" denotes the first academic semester of 2018). To maintain confidentiality, we have chosen to provide only the total number of distinct classes rather than their specific identifiers, as disclosing these could potentially allow for the identification of individual

instructors via the institutional website. The submissions listed in the table had passed all test cases. As the analysis for RQ2 did not consider submissions of assignments 1, 14, 15 and MC³ G4, the table also presents the totals after these exclusions.

Table 4.3: Distribution of classes, students, assignments, and submissions per academic terms analyzed in this research.

				Submissions	
Term	Classes	Students	Assignments	All	Used in RQ2*
2018-1	7	754	10	4,649	3,969
2018-2	7	657	12	5,087	4,505
2019-1	6	567	12	4,670	4,129
2019-2	8	623	12	4,896	4,315
2020-1	6	569	14	5,341	4,772
2022-2	7	634	15	5,811	5,076
2023-1	4	557	15	4,593	4,024
2023-2	8	625	15	5,835	4,930
Total	53	4,986	105	40,882	35,720

*Excludes assignments 1, 14, 15 and occurrences of MC³ G4.

Figure 4.1: Proportion of submissions per each analyzed term.

Figure 4.1 shows the proportion of submissions in the assignments of each term. The calculations were based on the "All" column in Table 4.3. These proportions serve to complement the subsequent analyses presented in this section by illustrating the trend of decreasing submissions throughout each term. To facilitate comparison, we identified the assignment with the highest number of submissions in each term and normalized the

submissions accordingly. This approach allows for a clearer understanding of the relative distribution of submissions across assignments. Overall, the figure highlights a notable decrease in submissions for later assignments compared to initial ones.

As described in Table 4.3, a total of 53 classes, covering submissions to 105 distinct assignments, were included in the analysis. As stated previously, MC102 is administered to most STEM programs within the university besides computer science and computer engineering. Consequently, classes in a single term comprised students from various programs. Considering this diversity, we argue that the substantial number of classes enabled a comprehensive large-scale analysis, encompassing students from distinct programs and educational backgrounds.

The total number of submissions for each term aligns with our expectations. According to the definition, the analysis of MC^3 is conducted solely on code that had passed all test cases of an autograder. However, in MC102, students' grades are determined by the number of tests their programs pass. Consequently, if students are already satisfied with a specific grade for an assignment, they may not invest additional effort to further elaborate their code to pass all test cases. This phenomenon also explains the decreasing number of submissions in later assignments, as illustrated in Figure 4.1. In these instances, the students had probably attained a minimum passing grade and chose to refrain from completing subsequent assignments or not to make additional efforts to pass all test cases.

4.5.2 RQ1: MC^3 frequency distribution

As detailed in Section 4.4, we aggregated assignments with equal numbers across the eight analyzed terms to facilitate analysis. The automated detection method that we used ensured that multiple occurrences of the same MC^3 within the same code were not double-counted. This allowed us to calculate the frequency distribution of each MC^3 relative to the total submissions for each assignment. Given the varying scales of these frequencies, we categorized them into three groups based on similar scales, as illustrated in Figures 4.2, 4.3, and 4.4.

Furthermore, it is important to note that the detection of certain MC^3 is based on predefined constants set by the instructor, as explained in Section 4.3. For our analysis, we established the following thresholds to determine the presence of specific MC^3: a constant value of 50 within the for loop (MC^3 C4), a total of 10 declared lists (MC^3 E2). These two were selected based upon expectations made upon the assignments' guidelines throughout the academic terms. Regarding the criteria for variable and function naming conventions (MC^3 G4), variables were required to have names with a minimum of four characters, functions with a minimum of eight characters, and no more than 70% of the total variables and functions could violate these naming requirements. The literature suggests that names should be meaningful; while some studies argue that longer names are beneficial [56], others highlight the potential impact of name length on human memory constraints [19]. To balance these perspectives, we set longer requirements for function names than for variables. Additionally, the 70% threshold allows some flexibility, permitting the use of shorter names, such as those commonly employed in loop iterations, where shorter, more concise names tend to remain meaningful [17]. While we did not manually assess whether

the submissions had the MC³ G4, we decided to keep them in our plots to illustrate its distribution based upon employed this criterion used in automated detection.

In general, individual frequency distributions reveal that, in general, nearly all MC³ occurrences are present throughout the academic term of the CS1 course. Arguably, MC³ B6 and B9 are the only ones that show a decrease in occurrences at a certain point, suggesting a potential disappearance.

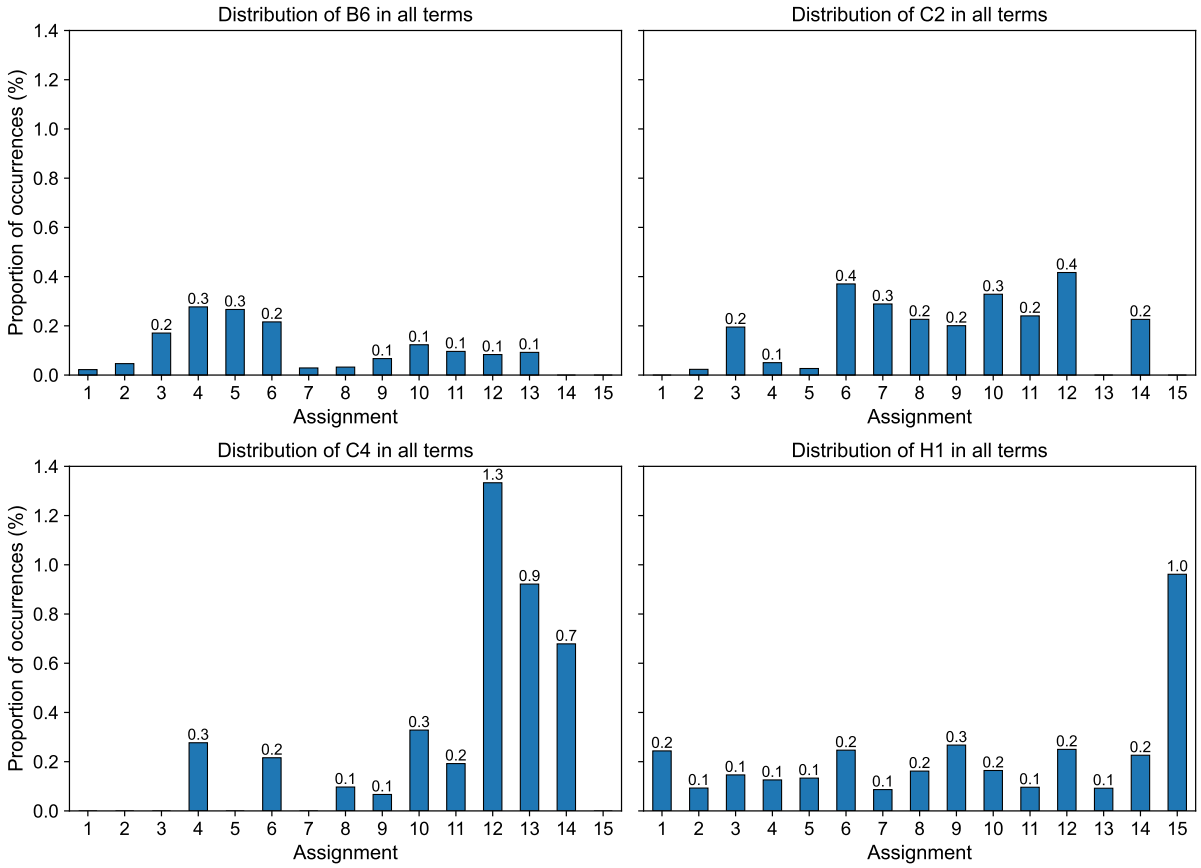

Figure 4.2: Grouped frequency distribution of MC³ B6, C2, C4, and H1. Ranges are between 0 and 1.5%.

The decision to calculate the frequency of each occurrence of MC³ (see Figures 4.2, 4.3, and 4.4) was based on two factors: normalizing the submissions with MC³ over the total submissions per assignment and aggregating data from the eight terms analyzed. Using the first factor, we aim to identify trends in each event, particularly focusing on assessing whether any MC³ disappears or decreases over the term. Regarding the second factor, we aggregated terms with varying total numbers of assignments. However, we argue that this decision had minimal influence on the results, as the MC102 syllabus remained consistent throughout these semesters, and the assignment order followed the syllabus in all terms. Thus, the overall trend would probably have remained unchanged regardless of the term aggregation approach.

Figure 4.2 illustrates the MC³ frequencies of the categories of Boolean expressions (B6), iteration (C2 and C4), and Other (H1). This figure grouped MC³ with the least proportion of occurrences, ranging from 0 to 1.5%. Despite their small proportions, these MC³

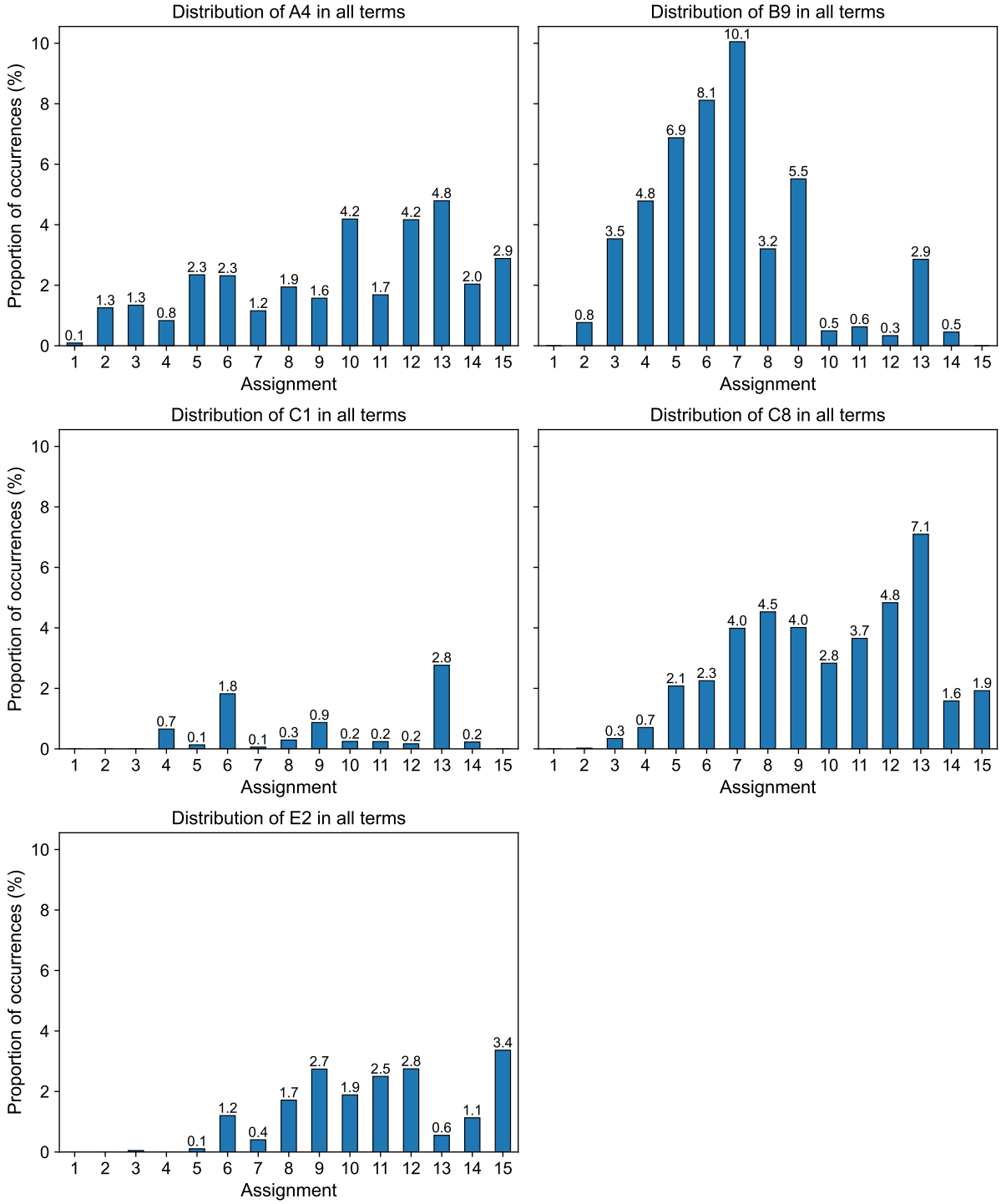

Figure 4.3: Grouped frequency distribution of MC^3 A4, B9, C1, C8, and E2. Ranges are between 0 and 10%.

persisted throughout the entire course. Confusion between `while` and `if` statements (B6) appeared to begin early and persisted until the 13th assignment, typically associated with sorting and selection algorithms. Similarly, C2, which involves iteration loops declared to execute only once, exhibited a similar persistence pattern. In contrast, C4 experienced a spike in the 12th assignment, expected to cover matrices. This spike suggested that

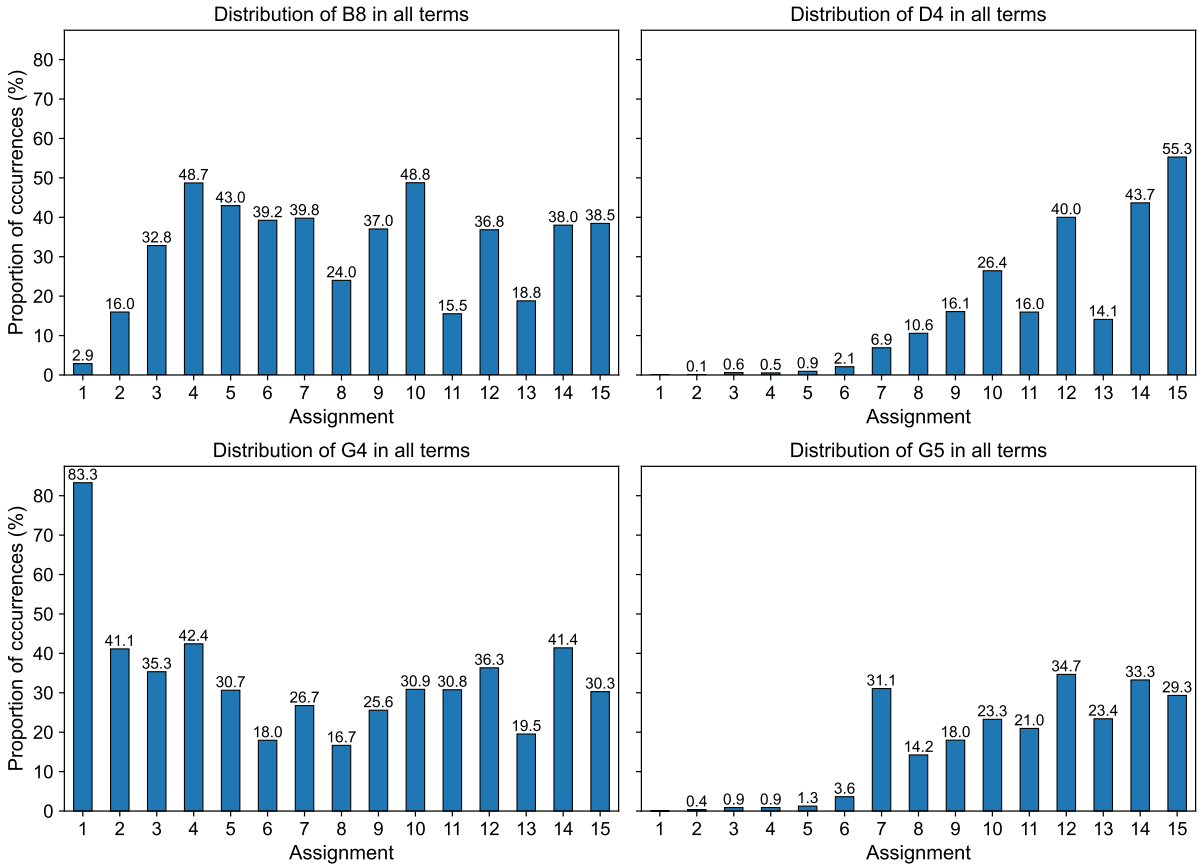

Figure 4.4: Grouped frequency distribution of MC³ B8, D4, G4, and G5. Ranges are between 0 and 80%.

students still struggle to understand the differences between `for` and `while` loops in the latter stages of the course. Furthermore, the results indicate that students declare statements that have no effect throughout the course (H1).

MC³ with proportions of occurrences ranging from 0 to 10% were grouped in Figure 4.3. Four distinct categories were represented: Variables, identifiers and scope (A4); Boolean expressions (B9); Iteration (C1 and C8); and Reasoning (E2). Throughout the course, students continued to redefine the Python built-ins (A4), indicating a persistent misunderstanding of these concepts. Although one might argue that novices may not initially be familiar with all built-ins, the results suggest that this misconception persists until the end of the course. The confusion about the concepts of `elif` (B9) exhibited a peculiar trend: its occurrence increased until it reached its peak in the seventh assignment, typically related to lists and tuples, and then decreased rapidly. This suggests that students may have learned to avoid this misconception only toward the end of the course. The incidences of C1 and C8 followed similar trends, persisting from early assignments until the course's conclusion. Both occurrences indicate a continued struggle with concepts related to `while` and `for` loops. In particular, MC³ C8 (`for` loop having its iteration variable overwritten) deserves further attention, as it was classified as the most severe in the work conducted by Silva et al. [119]. Regarding E2, the threshold of more than 10 user-declared lists used to determine this MC³ in a submission suggests that a fraction

of students may excessively rely on these structures in their code. This overreliance is especially prominent in assignments in the second half of the course, including the 9th assignment, usually dedicated to dictionaries.

Figure 4.4 presents the occurrences of MC³, with proportions ranging from 0 to 80%. Three distinct categories were identified: Boolean expressions (B8); Function parameter use and scope (D4); and Code organization (G4 and G5). Analysis revealed that students consistently declared `if-elif` sequences without an `else` throughout the course, making this MC³ the most frequent within its category. In contrast, D4 began to increase approximately midway until the end of the course, corresponding to the period when functions are introduced into the syllabus. This means that despite recommendations against the use of global variables, students continued to increase their use. Regarding G4, although we did not manually assess these submissions to verify the presence of variables or functions with non significant names, it is inferred that approximately 40% of students employed names violating the established minimum lengths we defined (four and eight characters, respectively). This observation was particularly notable in the 1st assignment, which provided guided solutions suggesting names that violated our rules. Lastly, G5 exhibited a similar trend to D4, as both MC³ involve functions. However, the presence of G5 suggests that the students did not consistently follow suggestions to declare their functions at the beginning of the code, demonstrating a lack of attention to code organization.

The frequencies of MC³ suggest that a fraction of students continue to struggle to grasp the fundamentals related to control and repetition commands by the end of the course. Furthermore, students exhibit a lack of attention or organization in their coding processes, as evidenced by the presence of MC³ A4, G5, and H1. Although our course analysis did not span the entire academic year, some distributions were similar to those observed by Altadmri and Brown [5]. The authors justified peaks in error frequencies with the introduction of new concepts in the course. In our analysis, the peaks were not necessarily when students learned a new concept, we argue that it was when they most needed to apply the concept related to each MC³. Furthermore, certain MC³ were significantly more frequent than others. Despite the type of error analyzed between Pritchard [104] and ours, the authors highlighted that errors with arbitrarily small frequencies are not uncommon. We posit that this is the same case with respect to MC³.

4.5.3 RQ2: Influence of the passage of time in MC³ occurrence

The results from RQ1 showed that, although the frequency distribution of each MC³ varies, they tend to appear at some point during the academic term and is potentially *persistent* until the end. Based on this evidence and also considering the flow of the academic term in terms of sequential assignments, the persistence analysis was structured as a comparison of MC³ occurrences between the first and second halves of the academic term.

As described in Section 4.4, our analysis focused on general MC³ occurrences. Based on that, we grouped assignments across all academic terms by their number and checked for occurrence of any MC³. However, the results in Figure 4.1 revealed a significantly low number of submissions for assignments 14 and 15 (around 20% of the total). Given the

potential impact of these low submission counts on the analysis, we also chose to exclude assignments 14 and 15. Consequently, the persistence analysis of MC³ was conducted using assignments 2 to 13, excluding MC³ G4, as previously explained.

The remaining 12 assignments (2 to 13) were divided into two groups to represent the first and second halves of the academic term: assignments 2 through 7 for the first half, and 8 through 13 for the second. For each assignment, we calculated the aggregated proportions of MC³ occurrences. Since the results did not meet the assumptions for parametric testing, we applied the Kruskal-Wallis test to assess the persistence of MC³ occurrences. The test results are presented in Table 4.4.

Table 4.4: Results obtained from Kruskal-Wallis test to compare MC³ occurrences between the first and second halves of MC102 academic terms.

Aggregated MC³ occurrences per assignment in all terms (%)	
1st half (2 to 7)	2nd half (8 to 13)
16.9	42.6
35.7	57.9
52.3	70.8
48.2	41.0
48.3	77.2
64.0	51.0
H	1.256
p-value	0.26
ε^2	0.02

The null hypothesis for this test posited that there were no differences between the two halves of the academic term. Rejecting the null hypothesis would suggest a significant difference in MC³ occurrences between the two halves. However, as shown in Table 4.4 and considering a significance level of 0.05, we failed to reject the null hypothesis, indicating no significant differences between the two groups. Furthermore, the small effect size reinforces the similarity between the halves in terms of MC³ proportions, supporting the view that these errors are stable or persistent throughout the academic term.

As for the analysis regarding proximity to assignments' deadlines, Figure 4.5 illustrates the distribution of MC³ occurrences in relation to the number of days remaining until assignment deadlines. We chose to truncate the results at the 21-day mark due to variations in the duration of academic terms. MC³ occurrences were calculated as the proportion of submissions containing at least one MC³ (except for G4) relative to the total number of submissions made each day, considering all assignments for that academic term. Although there are small variations per academic term, the results indicate that there is no discernible difference in the occurrence of MC³ based on the proximity of assignment deadlines.

As indicated by the results from RQ1, MC³ occurrences appear to be closely linked to the points in the course when students are most required to apply the concepts they have learned. As the course progresses, students must increasingly rely on these concepts in subsequent assignments, which explains the alternating but overall increasing cycle of

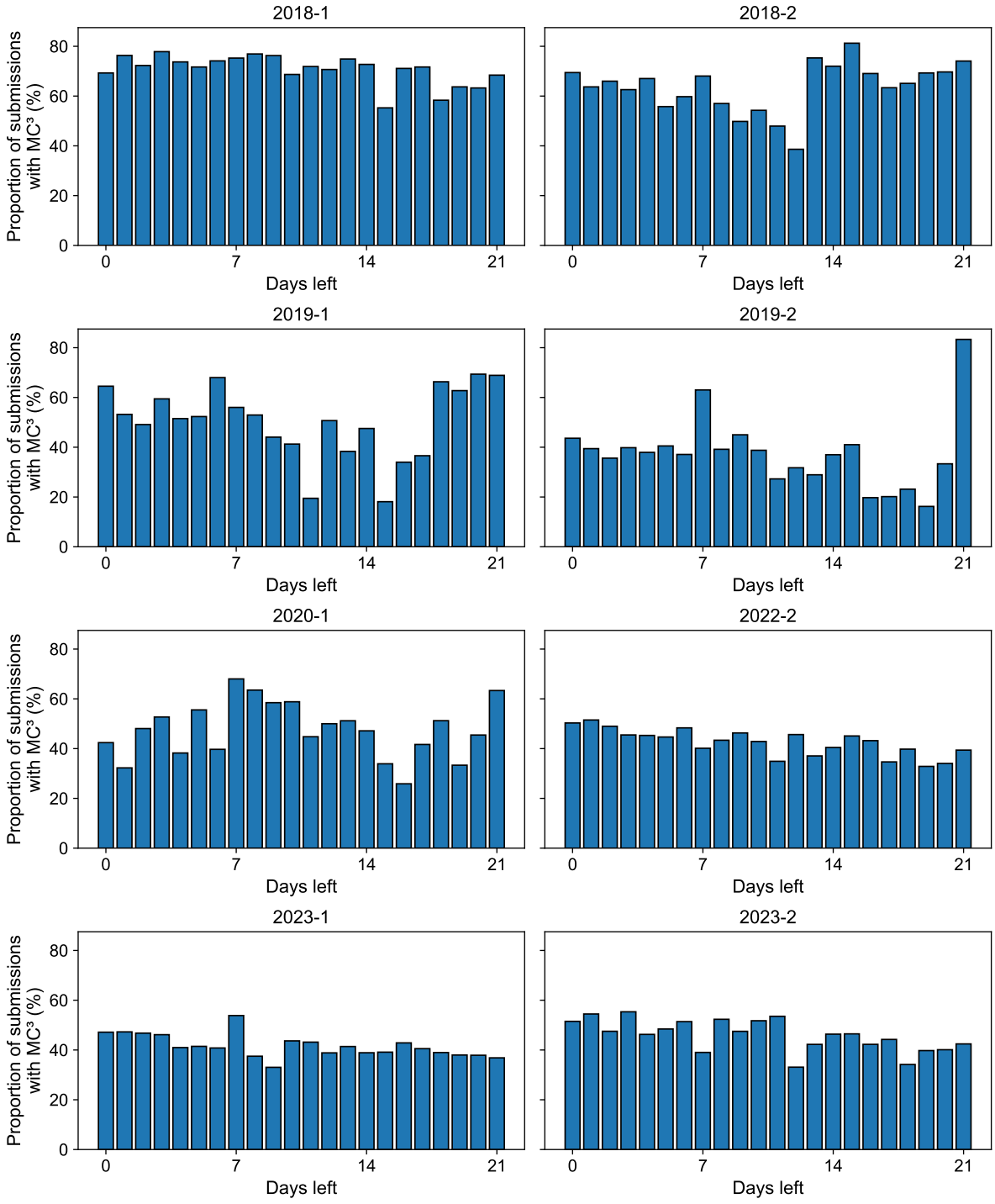

Figure 4.5: Proportion of MC³ occurrences per days left until assignments' deadlines.

occurrences shown in Table 4.4 and Figures 4.2, 4.3 and 4.4. The test results suggest that these frequencies are persistent throughout the academic term. This implies that MC³ occurrences begin at some point during the course and continue until its conclusion, indicating that students may even carry these misconceptions over into future CS courses. This persistence is concerning because, as noted in this study's motivation, focusing solely

on passing test cases through automated assessment is insufficient to address deeper issues like MC³.

On another aspect of the passage of time, the results from Figure 4.5 reveal that, contrary to our initial beliefs, the proximity of the assignment deadline did not seem to influence the occurrence of MC³. Our hypothesis posited that students, when under time pressure near the assignment deadline, might prioritize correctness over other aspects of their code. Although we did observe a generalized greater number of submissions on the last day, normalization of MC³ occurrences in proportion to all submissions indicated a relatively stable pattern over the 21-day period analyzed. These results suggest that factors other than the submission date may contribute to the occurrence of MC³.

To further examine the distribution of aggregated MC³ occurrences presented in Table 4.4, an autocorrelation test was conducted. Our objective was to identify potential trends in MC³ occurrences over the course of the academic term. Although the test results (omitted in this study) indicated cyclic patterns, all values fell within the confidence interval, suggesting no significant evidence of a lagged relationship between MC³ occurrences across assignments. In other words, while the Kruskal-Wallis test indicated persistence of MC³, this persistence does not appear to relate from occurrences in past assignments. This supports the idea that other factors, such as broader learning difficulties with specific concepts (as discussed in Section 4.3), are the underlying causes for MC³ occurrence.

4.6 Implications for CS1 teaching practices

The analysis from both RQ1 and RQ2 conducted an investigation of MC³ occurrences both individually and in general. The varying frequency distribution highlights the unique nature of each MC³, potentially linked to the specific topic and the difficulty involved in correcting it. We recognize that misconceptions like built-in redefinition (MC³ A4) and overwriting the `for` loop iteration variable (MC³ C8) arise from different cognitive processes and, as such, may require distinct approaches for rectification.

Students' emphasis to code correctness when using autograders has a significant impact on the occurrence of MC³. According to the Dual Process Theory [108], two complementary cognitive systems exist: the intuitive and the reflective. When students focus solely on passing tests, they primarily engage their intuitive system, which leads them to overlook other code characteristics, such as significant variable or function naming (MC³ G4), code organization (MC³ G5), or built-in redefinition (MC³ A4). Another consequence is related to the perceived cost [143] of writing code while paying attention to these characteristics. The lack of evaluation beyond correctness may cause students to neglect these aspects, as there is no immediate reward for addressing them. We advocate that this neglect could also contribute to the development of MC³.

As noted by Joni and Soloway [70], focusing on code efficiency in terms of runtime and resource usage is often not persuasive enough for novices to appreciate code characteristics beyond correctness. This poses a significant challenge in encouraging CS1 students to recognize and address MC³. Instructors and teaching assistants should seize opportunities during lectures to demonstrate real-life examples of how poor readability

and maintainability can negatively impact code. However, it is important to acknowledge that student programming skills are not solely shaped within the confines of CS1 classrooms. The presence of habits developed in previous programming knowledge, access to external educational materials, and peer interactions also play significant roles in influencing students' coding behaviors. Therefore, supplementary educational materials that highlight these differences, along with the integration of automated feedback on MC³ into autograder systems, can further support students outside the classroom.

4.7 Limitations and threats to validity

The main limitation of this study is the reliance on data from a single higher education institution, which offered a CS1 course taught in Python using the imperative paradigm. To mitigate this limitation, we focus on the coordinated environment of the CS1 course, which enrolls students from various STEM bachelor's programs each term. Although this allowed us to study a diverse cohort of undergraduates with different educational backgrounds, it is important to recognize that the results may vary if data from other institutions were analyzed. Furthermore, we acknowledge that the replication of this study is simplified due to the popularity of Python as a language in CS1 [58], and the fact that the syllabus of MC102 covers most of the topics addressed in these courses [118].

A potential threat to validity stems from the use of our MC³ automated detection tool, given the limitations and challenges encountered during its development. To address concerns regarding the instructor's expertise, we reviewed the assignment details and selected appropriate threshold values for the required metrics used in automated detection (refer to Section 4.3), such as the maximum expected iterations for `for` loops (MC³ C4) and the total number of user-declared lists (MC³ E2). Additionally, we opted to exclude G4 from the persistence analysis due to the impracticality of accurately assessing this MC³. The tool has been made publicly available, allowing other researchers to utilize and enhance it.

4.8 Conclusions

This research aimed to assess occurrence's behaviors of misconceptions, referred to as Misconceptions in Correct Code (MC³), that programming novices exhibit in code that yields the expected outcome. These misconceptions potentially indicate that students have poor or incomplete understanding of the CS1 topics. To achieve our objectives, we conducted a large-scale study analyzing over 40,000 student submissions from eight distinct terms of a CS1 course.

The study revealed that, even at the end of the CS1 course, a fraction of students appeared to lack complete comprehension of concepts related to conditional and repetition commands. In addition, students also exhibited a tendency to neglect the readability and maintainability of their code. Evidence suggests persistence in MC³ occurrence, meaning that students start developing MC³ at certain points during the CS1 course and carries them until its end, possibly even carrying them to future CS courses. Additionally, the

occurrence of MC³ does not seem to be influenced by time constraints of assignments' deadlines nor is dependent on previous occurrences, further suggesting that the underlying causes are indeed rooted in learning difficulties to specific CS1 concepts.

The research proposed in this paper serves as a cautionary note to CS1 instructors that students continue to incorporate MC³ into their code by the end of the course. Although our studies focused on Python courses, it is not unfathomable that MC³ can also occur in other programming languages [119]. Our results contribute to pave the way for future studies that aim to address MC³, enriching discussions among peer researchers by highlighting our successful and unsuccessful approaches [136]. Furthermore, we acknowledge and advocate for the adoption of rather new technologies, such as enhanced AI-generated feedback, which holds promise in assisting both instructors and students in the CS1 teaching and learning process. This integration of traditional and novel educational tools bridges a gap in empirical research by leveraging existing results alongside new ones [85].

4.9 Afterword: a Case Study to assess educational materials

This afterword serves to complement Chapter 4 by providing additional data that was omitted from the original article due to space constraints. The primary contribution of this section is a case study evaluating the educational materials developed for MC³, which include lecture slides, short videos, and flashcards. The study aimed to address the following research question:

- **RQ**: In what ways can educational materials be used to teach CS1 students about MC³?

The case study was conducted with 23 students from a MC102 class taught in the second academic term of 2023. The results suggest that the use of specific educational materials can have a positive impact on mitigating MC³. However, the findings also highlight challenges related to the implementation of these materials and student engagement. By demonstrating the persistence of MC³ and the positive impact that short videos, flashcards, and slides can have on mitigating these misconceptions, the studies presented in this chapter lay the groundwork for the development of additional educational materials to address MC³ in CS1 courses.

4.9.1 Related work on addressing misconceptions

Plass-Oude Bos [100] conducted a study involving 41 secondary and undergraduate students to identify and rectify common misconceptions related to variable assignment and manipulation. The study focused on six learning goals, for which an interactive educational video was developed to assist students in understanding the correct concepts. Multiple-choice questions embedded within the video provided interactive engagement with the students. The video adhered to suggested guidelines [137] on length, conciseness,

and clarity of explanations for presented tasks. The pre and post-tests were administered as questionnaires to evaluate student understanding, and Plass-Oude Bos reported a significant improvement in students' performance on the post-tests, indicating a positive impact of the educational video.

The study conducted by Plass-Oude Bos [100] was a collaborative effort with Peters [97], who provided further information on the effectiveness of the educational videos developed by Plass-Oude Bos. Among the details explained by Peters was the assessment of the videos' effectiveness in addressing misconceptions and instructing learning goals related to variables in imperative programming. This effectiveness was evaluated based on the number of incorrect responses to a post-test questionnaire that assessed the students' understanding after watching the videos. Peters reported that some misconceptions were fully resolved, meaning that they no longer occurred. Others varied significantly and marginally diminishing in terms of recurrence, and one misconception increased its occurrence after viewing the video. Regarding learning goals, students demonstrated a better understanding of variable names, assignment structures, storage limitations, and variable changeability.

Multimedia artifacts have been used to address misconceptions in other aspects of computer science education. In their work, Blank et al. [20] presented experimental results from a multimedia course designed to correct misconceptions about computer science of students, which could negatively impact their perception of the field. The experiment involved 55 undergraduate students who completed pre- and post-test questionnaires assessing their opinions on stereotypical beliefs about computer science (e.g.,"a strong math background is needed to succeed in computer science"). The authors reported that the multimedia course successfully countered these negative stereotypes, leading to positive changes in students' perceptions of computer science.

Chiodini et al. [36] compiled a curated inventory[3] of programming misconceptions with the primary aim of providing students and educators with easy access to these types of resources. The inventory resulted from assessments of observations, programming activities, and artifacts produced by students across various courses over a decade. The inventory initially targeted Java, but was expanded with Python, JavaScript, and Scratch. Organized by concept, programming language, language specification, and textbook section, the inventory contains 198 misconceptions, each described by several attributes, including explanation videos. A central feature of the inventory is the juxtaposition of the wrong aspects in the misconception with the explanation of how the correct form should be. This approach is known as *refutation text* and has the objective of facilitating notions behind conceptual changes [132]. The inventory is hosted at `progmiscon.org`.

Gomes et al. [55] created an educational artifact comprising multiple-choice questions designed to aid students and instructors in introductory programming courses. This artifact is grounded in the principles of programming antipatterns [27], which are commonly encountered solutions to problems resulting in negative consequences. The authors developed a total of 63 questions for C and 46 for Python, with the difficulty of the questions tailored to the associated antipatterns. Additionally, Gomes et al. conducted a pilot test to evaluate the correctness of the questionnaire. While the authors expressed overall

[3] `progmiscon.org`

satisfaction with the results, they noted that the small number of participants warrants further investigation.

Evans et al. [47] developed SIDE-lib, focusing on identifying symptoms and misconceptions in Python code. Symptoms, defined as small parts of code that reveal misunderstandings of programming concepts, can lead to correct or incorrect code, indicating underlying misconceptions held by the programmer. The authors obtained symptoms and misconceptions by analyzing 1,331 submissions from students. SIDE-lib can detect 33 symptoms and 25 potential misconceptions. Outside the scope of introductory programming courses, Ferrao et al. [48] developed embedded-check, a tool for detecting high-level conceptual mistakes in an Embedded Systems course. This tool, elaborated as the result of several years of course observation, detects 13 designed rules that indicate conceptual errors. Ferrao et al. highlighted that the embedded check enables timely formative feedback to students and reduces the workload of instructors in assessing assignments.

4.9.2 Methods

Building on insights obtained with respect to the frequency analysis of MC³, described in Section 4.5, we developed a set of educational materials focused on the nine most frequent MC³. These materials aimed to revisit concepts taught in the CS1 course while explaining why code should not be structured with MC³ and providing strategies to avoid such coding behaviors, similar to refutation text approach used by Chiodini et al. [36]. The educational materials can be consulted in Appendix B of this thesis.

We chose to develop the educational materials in the form of short videos (under 60 seconds, based in the format of YouTube Shorts[4]) and flashcards, anticipating that these formats would be more accessible to students. The use of these educational materials also aligns with the principles of spaced repetition [22, 43], which helps reactivate students' retention of knowledge. The learning of new concepts follow a "forgetting curve" [22, 96, 113], indicating that the concepts that are learned tend to be forgotten unless they are revisited periodically. The theory suggests that the time required for reactivation decreases over time, making the content easier to remember and eventually leading to peak retention. Additionally, previous studies in CS1 classes have shown positive results with the use of videos, either as supplementary instructional material [129] or as the primary material [87]. Additionally, flashcards are commonly utilized in Active Learning techniques [22], with demonstrated positive impacts observed in both basic education [2] and higher education [80, 109, 150].

Given the context of this study, a case study emerged as the optimal research method. Case studies are frequently employed when researchers have limited control over events [37], as anticipated in our scenario where the educational materials would be introduced into an ongoing CS1 class comprising students with diverse backgrounds. Moreover, case studies focus on individual actors and highlight specific events relevant to the case [37], aligning with our focus on the identified MC³. Our case study can be classified as explanatory [37], as its objective was to assess whether the educational materials positively influenced students' reasoning in code development.

[4] `https://www.youtube.com/intl/en/creators/shorts/`

From the list of the 14 MC³ presented in Section 4.5, we selected the following for focus: A4 (redefinition of built-in), B8 (non utilization of `elif/else` statement), B9 (`elif/else` retesting already checked conditions), C1 (`while` condition tested again inside its block), C8 (`for` loop having its iteration variable overwritten), D4 (function accessing variables from outer scope), E2 (redundant or unnecessary use of lists), G4 (functions/variables with non significant name), and G5 (arbitrary organization of declarations). These MC³ appeared to be the most frequent. However, due to the complexity of representing MC³ B8, E2, and G4 in short videos and flashcards, we decided to create specific lecture slides to address them instead. Since this case study approach involved human participants, it received prior evaluation and approval by an Ethics Research Committee affiliated with Universidade Estadual de Campinas[5].

4.9.2.1 Data collection

The case study was conducted during the second academic term of 2023. In this term, I acted as both instructor and researcher. Students were volunteered for the study during the first two weeks of the course. Following recruitment, students who agreed to participate via an Informed Consent Form were divided into two groups: Group A and Group B. The division process involved ordering students by their ID numbers and then alternating the allocation of students between the two groups.

Group A received short videos and flashcards related to MC³ A4, C1, and D4, while Group B received for MC³ B9, C8, and G5. These resources were disseminated via institutional email to students after the corresponding CS1 topics were taught and before the related assignments were released. This process envisioned the spaced repetition while also providing new information about MC³. The videos were uploaded in YouTube while the flashcards were created digitally using Canva[6] and attached to the email. Both groups received one video and one flashcard simultaneously, ensuring balance between the groups; for instance, while Group A received materials for A4, Group B received materials for B9, and so forth.

Additionally, the slides covering MC³ B8, E2, and G4 were presented in lectures to all attending students, regardless of their participation as volunteers, following the same principle of being taught after the corresponding CS1 topics were covered. However, non-participants did not have their code evaluated.

In this case study, our hypothesis was that students in each group would potentially incorporate less of each MC³ after they received instruction about them. Additionally, we anticipated low occurrences of MC³ B8, E2, and G4 for both groups, since these were taught in lectures. To optimize results, students were encouraged to review the materials before attempting the assignment. Students also were requested to refrain from sharing the videos and flashcards. However, it was not possible to ensure compliance with the latter requirement.

Although the academic term comprised a total of 15 assignments, the intervention period extended until the deadline for the tenth assignment. During this time, submis-

[5]Approval can be consulted in Plataforma Brasil with CAAE number 70220523.0.0000.5404.

[6]`https://www.canva.com/`

sions from participating students were collected via the institutional autograder after the deadline for each assignment. Additionally, students were administered two electronic questionnaires during the intervention period: one midway and another at the end. The second was only sent to respondents of the first one. These questionnaires aimed to determine when students watched each short video, flashcard, and MC^3 lecture slides presented. They also included questions soliciting students' opinions on the developed materials.

4.9.2.2 Data analysis

The MC^3 frequency distribution was calculated using the developed automated detection tool. However, given the focus on evaluating the educational materials, the primary objective was to assess occurrences before and after students viewed each dedicated MC^3 material. Based on the assignment number reported by students in the questionnaires, valid submissions (i.e. submissions that passed all test cases) for an assignment were classified into one of four cases:

- **O-W**: Submissions with an occurrence of MC^3 X where the student had watched the materials related to X.
- **O-NW**: Submissions with an occurrence of MC^3 X where the student had not watched the materials related to X.
- **NO-W**: Submissions with no MC^3 occurrences where the student had watched previous MC^3 materials.
- **NO-NW**: Submissions with no MC^3 occurrences where the student had not watched previous MC^3 materials.

The first assignment was discarded in this analysis since it had a guided solution. Regarding MC^3 G4, I manually assessed its presence in submissions from Groups A and B. If a submission contained more than one MC^3 and the student had watched materials related to at least one of them, it was classified as O-W. Submissions for each group were categorized separately, ensuring that specific MC^3 taught to one group were not counted as occurrences for the other group.

On the other hand, for MC^3 B8, E2, and G4, which were taught in lectures, we determined that all assignments after that lecture would consider the corresponding MC^3 as covered. Consequently, all assignments were classified as either O-NW or NO-NW before the lecture and as O-W or NO-W after the lecture.

Students who did not respond to the first questionnaire were excluded from the research. If a student only answered the first questionnaire, their responses were considered, but any subsequent sent educational materials were automatically classified as not watched by the student. Additionally, submissions that did not pass all the test cases were deemed invalid and therefore discarded from analysis.

All occurrences classified as O-W, O-NW, NO-W, and NO-NW were tallied and organized into one contingency table for Group A and another for Group B. The following

conditions indicated that the materials had a positive impact on reducing MC³ occurrences:

1. The number of occurrences where students had watched the materials was lower than those where they had not watched (O-W < O-NW).

2. The number of submissions with no MC³ occurrences where students had watched the materials was greater than those where they had not watched (NO-W > NO-NW).

3. If both conditions 1 and 2 were met, we assessed whether the frequency distribution of the contingency table significantly differed from one obtained by chance. To accomplish this, the Chi-Squared test [81] was employed. Further details regarding the hypothesis can be found in Section 4.9.3.

The other questions regarding students' opinions on the educational materials encompassed various field formats. Likert-items responses were analyzed using descriptive statistics, while responses from open-text fields were subjected to content analysis [81]. Specifically on the latter, the aim was to identify students' perspectives on the integration of these materials into CS1 lectures and the MC³ they had learned to avoid incorporating in their code.

4.9.3 Results and discussion

A total of 28 students volunteered to participate in the case study research, resulting in Groups A and B consisting of 14 students each initially. However, due to reasons such as abandonment of the course or failure to respond to the first questionnaire, the analysis was based on data from 12 students in Group A and 11 students in Group B. The following paragraphs show the findings on MC³ occurrences and students' opinions on the educational materials separately.

4.9.3.1 Students' interaction with the materials

Table 4.5 displays the overall interaction the students had with each MC³ educational material. As previously explained, the short videos and flashcards were sent simultaneously by the researcher, and both groups received their assigned materials at the same time. In addition, the table presents the total number of students who had watched each material. These totals were calculated based upon students' answers to the administered questionnaires.

Overall, engagement with V&F 1 and 2 was satisfactory, as most of the students in both groups indicated that they had watched the materials. Students' responses to when they had watched the materials varied, but when calculating median and mode values to these responses, the results confirmed that the students did indeed access both videos and flashcards near the dedicated assignment. However, the initial statistics obtained from YouTube's dashboard suggested that the videos were not being viewed as expected. In response, additional efforts were made to increase student participation, including

Table 4.5: Description of students' interaction with each developed educational material.

Materials	MC³		Total of students who watched each material	
	Group A	Group B	Group A (N = 12)	Group B (N = 11)
V&F 1	A4	B9	12	11
V&F 2	C1	C8	12	11
V&F 3	D4	G5	7	10
Slides 1	G4		7	9
Slides 2	B8		5	6
Slides 3	E2		4	7

reminders to watch the materials. It is plausible that this constant reminder process contributed to improve engagement among students.

The slides, on the other hand, apparently did not receive significant engagement. We argue that the main reason for this was the optional lecture attendance policy implemented in the second term of 2023, period in which the case study was conducted. Despite the researcher uploading slides to Google Classroom after each session, it is possible that students simply ignored them. In addition, a strike affected teaching across the entire institution where the case study was conducted, resulting in almost four weeks without lectures. We propose that the strike could have influenced overall engagement, particularly regarding students' engagement with V&F 3, as depicted in Table 4.5.

4.9.3.2 MC³ occurrences in Groups A and B

Taking into account the nine assignments that comprise the intervention period, Group A was expected to produce a total of 108 submissions, while Group B was expected to yield 99 submissions. However, after excluding invalid submissions (i.e., those that did not pass all test cases), the analysis was conducted with 93 submissions from Group A and 94 from Group B. The contingency tables summarizing the frequencies of O-W (occurrences when students had watched), O-NW (occurrences when students had not watched), NO-W (non-occurrences when students had watched), and NO-NW (non-occurrences when students had not watched) are presented in Tables 4.6 and 4.7 for Group A and Group B, respectively.

Table 4.6: Contingency table for Group A.

	NW	**W**
O	30	26
NO	10	27
χ		5.4
p-value		0.021

Table 4.7: Contingency table for Group B.

	NW	**W**
O	13	39
NO	8	34
χ		0.2
p-value		0.660

Since the cells in both contingency tables were greater than five and represented mutually exclusive scenarios, such as a submission being classified as either O-W or O-NW,

the Chi-Squared test was employed. The null hypothesis posited that the distributions of rows and columns were obtained by chance. For Group A, the test resulted in $\chi = 5.4$ and p-value = 0.021. In contrast, for Group B, the test resulted in $\chi = 0.2$ and p-value = 0.660. With a significance level of 0.05, we rejected the null hypothesis for Group A but not for Group B. The results obtained from Group A met all the established conditions outlined in Section 4.9.2 (O-W < O-NW, NO-W > NO-NW, p-value < 0.05). However, the results of Group B only satisfied the second condition (NO-W > NO-NW).

The results presented in Tables 4.6 and 4.7 depict lessons learned and new opportunities. First, the small values in NO-NW for both groups indicate that MC^3 are not randomly incorporated by the students, which further supports that MC^3 are persistent if they are not addressed in class. Secondly, despite the apparent low engagement with the materials, Group A met all the established conditions to suggest a positive impact in mitigating the occurrences of MC^3. In contrast, Group B had the O-W tripled compared to O-NW (39 and 13, respectively). Although one might be tempted to argue that the materials had the opposite desired outcome for Group B, since we could not reject the null hypothesis for this group, we cannot consider its results as strong as those for Group A. Furthermore, different mitigation effects in MC^3 could explain the variations between O-W and NO-W, as similar effects were reported by Peters [97] regarding students who watched the interactive video.

4.9.3.3 Students' responses to the questionnaires

The first questionnaire was administered to students after the lecture and presented slides that addressed MC^3 E2. At this moment, the students had received materials V&F 1 and 2, as well as slides 1 and 2 (see Table 4.5). In total, the first questionnaire collected responses from 12 students in Group A and 11 students in Group B. Subsequently, the second questionnaire was distributed immediately after the end of the intervention period. At this time, Group A had seven respondents for the second questionnaire, while Group B had ten respondents.

When asked with the statement "the videos and flashcards had an appropriate didactic language," all 17 students classified it as either "agree" or "strongly agree". These opinions were also reflected in their perspectives on the incorporation of these materials into the CS1 course. In Group A, four students advocated for the integration of both videos and flashcards, while one student suggested that only flashcards should be integrated, and two students believed that the materials should not be part of the lectures. Similarly, in Group B, seven students supported the integration of both materials, one student preferred only flashcards, and two students opposed the inclusion of the materials in lectures. Students who favored both materials or just the flashcards argued that integration would not consume excessive lecture time, while those who disagreed expressed skepticism about the fit of these materials in a standard lecture setting.

Although we analyzed responses to questions about the MC^3 students learned to avoid, we chose to omit details of these results. The main reason for this was that the majority of students in both groups stated positive impacts in their learning, but as discussed earlier, evidence suggested positive impacts only for Group A. However, a few contrary

opinions could shed light on this outcome. Two students expressed the view that videos and flashcards alone were not sufficient to understand MC³, emphasizing the need for in-person explanations of the topics. Additionally, another student suggested incorporating a live code compilation to illustrate the differences between code with and without MC³. This perspective raises concerns about a potential misunderstanding of MC³, as the code output would remain unchanged regardless of their presence.

Despite the low response rate to the questionnaire, students expressed positive feedback on the editing, formatting, and language used in the educational materials. Regarding the videos, we attribute this positive opinion to the format of YouTube Shorts. The constraint of a 60-second length and the vertical aspect influenced the production of our videos. In line with the refutation text approach for explaining MC³, our videos met criteria such as length and conciseness, suggested by previous studies on video-based content for CS1 [36, 87, 97]. We posit that these videos can be further refined and utilized in future iterations of MC102, aligning with the reusability value emphasized by Stephenson [129]. Similarly, the development of flashcards followed a digital format to facilitate student access. Most of the respondents considered flashcards to be complementary to the videos. Although evidence suggests that this type of media is popular among college students [109], the results obtained in this case study further contribute to the belief that the full pedagogical potential of digital flashcards remains to be fully explored [150].

4.9.3.4 Limitations and threats to validity

The main limitation of this study stems from it being conducted in a single academic term of a higher education institution. Moreover, the obtained results are significantly dependent on the number of participants, which was low. Although we observed small positive effects in Group A, caution should be exercised in generalizing these findings. To mitigate this limitation, comprehensive details of the methodology were provided while also highlighting potential avenues for future research iterations.

4.9.4 Conclusions

This afterword presented a case study designed to assess educational materials developed to teach students about MC³. A total of 23 students participated in the study, which was conducted in a MC102 class taught in the second academic term of 2023 at Universidade Estadual de Campinas. Since the study involved human participants, it received prior evaluation and approval by the Ethics Research Committee of the same institution.

The obtained results indicated that explanation-driven short videos, flashcards, and slides had a positive impact on MC³ mitigation. However, these educational materials were not sufficient to achieve the desired outcome. Evidence suggests that although these formats initially appealed to students, they were not prominently accessed throughout the CS1 course. This lack of consultation may be attributed to automated grading systems that primarily assess code correctness, thereby reinforcing students' focus solely on this aspect. However, I agree with the assertions made by Joni and Soloway [70], advocating for instructors to provide feedback based on the concepts of readability and maintainability rather than solely on code efficiency when addressing MC³ to novice programmers. This

presents a challenge, as instructors may find themselves in the dilemma of effectively convincing students that their already working solutions need to be penalized, but, at the same time, instructors should not forget that the purpose of assessment is to teach and not to grade [45].

Chapter 5

Discussion

This chapter endeavors to synthesize the findings of this thesis. While Chapters 2 to 4 provided a clear chronological progression of this work, certain details may have been dispersed across these chapters. Given that each chapter has addressed its respective research questions, I have chosen to structure this discussion by revisiting each specific objective defined in Chapter 1.

This thesis did not fully explore all 45 identified MC^3 listed initially; instead, the focus was directed towards the 15 MC^3 classified as most severe. To aid readers in this chapter, Table 5.1 lists all MC^3 and each related teaching intervention developed in this thesis. All developed teaching interventions can be found in Appendix B. Recall that MC^3 were divided into eight categories: 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).

Insights into the reasons for the occurrence of other MC^3, which were not extensively explored, were gathered through conversations with MC102 students and consultations with CS1 instructors. To compile all information related to MC^3, including those not extensively covered, I have included a catalog in Appendix A. I chose to keep in the catalog all originally identified MC^3 to enable other researchers to recognize them in their teaching environments and develop educational materials based on this thesis' findings.

5.1 SO1: Identification, analysis, and validation of MC^3

This objective served as the central contribution of this thesis. The identification and analysis of MC^3 were accomplished through the study of students' code within a specific educational context. While this may be considered a limitation, it is not uncommon in similar research endeavors where the exploration of misconceptions began within specific environments before broader validation was pursued [6, 28, 36, 92]. As outlined in SO1, the validation of MC^3 required both internal and external assessments, which were conducted iteratively throughout the development of this thesis.

The internal validation process involved semi-structured observations within MC102 classes and analysis of automatically detectable MC^3 in students' code. Conversations

Table 5.1: List of MC[3] and their related teaching interventions developed in this thesis. The middle horizontal line denotes the 15 most severe MC[3]. Table sorted in lexicographical order by MC[3] ID.

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

DIF: Total difference between CS1 instructors who agreed MC[3] was severe and those who did not (Chapter 3).
AD: Automated Detection. **SV**: Short Video. **FC**: Flashcard. **LS**: Lecture Slides.

with MC102 students confirmed some of my initial expectations regarding incomplete or incorrect comprehensions of CS1 topics. However, I was particularly struck by students' explanations for MC[3] such as C1 (`while` condition tested again inside its block) and G5 (arbitrary organization of declarations), which were rooted in concerns related to the

autograder. Some students wanted to ensure the system would not wrongly assess their code, leading to the declaration of redundant checks. Others were overly cautious about plagiarism detection, organizing their code in (confusing or redundant) ways they believed would differ from typical student submissions. When curating the initial list of MC³, I had not anticipated that the use of an autograder could influence MC³ reasoning aside from incomplete or incorrect comprehensions.

Students' preference for one construct over another may contribute to the occurrence of MC³ such as C4 (arbitrary number of `for` loop instead of `while`) and E2 (redundant or unnecessary use of lists). The mere-exposure effect [148] can help explain this phenomenon. In CS1 context, students who are repeatedly exposed to exercise solutions that utilize a particular construct may become more familiar with it, leading them to prefer its use even when other constructs would be more appropriate.

The occurrence of some MC³ reflecting a careless approach to coding may also be linked to how students perceive the effort required to adhere to these practices within their educational context. Wigfield and Eccles [143] describe that the *cost* of engaging in a specific activity limits access to other activities based on the effort required for the former. This concept can be related to MC³ in the Code Organization category. When the CS1 course focus solely on code correctness, students may not understand the benefits of using meaningful variable or function names (MC³ G4) and maintaining organized code (MC³ G5). As a result, students may perceive the costs of adhering to these practices as too high since they would not understand the reason why these should be adhered to. This reinforces the importance of instructors actively engaging students by demonstrating practical scenarios in which obfuscated and disorganized code proves detrimental.

A lack of attention during programming was also considered as cause for MC³ A4 (redefinition of built-in) and G4 (functions/variables with non significant name). The Dual Process Theory in cognitive psychology [108] postulates the existence of two cognitive systems: one that is fast and intuitive, and another that is slow and reflective. The reflexive system is associated with rule learning, which can be linked with the awareness of coding behaviors such as MC³. When students focus primarily on passing all test cases in an autograder, they may engage the fast system, potentially neglecting attention to other important aspects of their code.

The initial frequency distribution of MC³ occurrence was presented in Chapter 3 and further explored in Chapter 4. As demonstrated, the 14 automatically detectable MC³ manifest at some point during the CS1 course and persist by the conclusion of the course. This permeates the notion that students can complete the course with top grades while retaining incomplete or incorrect understandings of the material. While the frequency varies for each MC³ (refer to Table 3.4 and Figures 4.2, 4.3, and 4.4), eight of these misconceptions relate to core concepts such as decision and iteration structures. The remaining MC³ reflect careless approach to coding. Despite arguments that novice programmers, especially from non-STEM backgrounds, should not prioritize these aspects [10, 51], I contend otherwise. The 15 most severe MC³ encompass either incomplete understandings of CS1 topics or coding behaviors that impact code readability and maintainability. Early emphasis on these skills should be integrated into CS1 courses, as programming is

typically a collaborative endeavor. On the other hand, I do believe that addressing code efficiency should only occur in subsequent programming courses after CS1.

The external validation was carried out by the questionnaire and semi-structured interviews with CS1 instructors, complemented by syllabi analysis in Chapter 2. Consultations with CS1 instructors not only allowed the identification of the most severe MC^3 but also provided insights into diverse teaching contexts beyond MC102. For instance, as discussed in Chapter 3, six instructors reported using autograders, but only three manually assess students' code even when it passes all test cases. This highlights additional teaching contexts where MC^3 may manifest, potentially allowing students to complete the CS1 course without addressing these misconceptions. Moreover, instructors who prefer manual assessment emphasize its efficacy in gauging students' grasp of course material, especially when assessing code with MC^3 like behaviors.

The syllabi analysis conducted in Chapter 2 yielded insights into CS1 teaching contexts in Brazil, revealing notable similarities to the MC102 course. For example, 90% of CS1 courses emphasize programming through the procedural paradigm. Even though C seemed to be the most popular language, Python was the second one. Furthermore, as discussed in Chapter 3, the most severe MC^3 can potentially happen across other programming languages, particularly those within categories B (Boolean expressions) and C (Iteration). Table 5.2 expands upon Table 2.10 by illustrating MC^3 categories that are related to the most covered topics identified in the syllabi analysis. MC^3 categories G (Code organization) and H (Other) were not included in the table, as the related misconceptions can arise from any topic covered in CS1. These findings illustrate the presence of diverse CS1 teaching contexts where students may develop MC^3. However, as further discussed in this chapter, I do not assert that these findings alone indicate that CS1 students develop MC^3.

Table 5.2: MC^3 categories related to the most covered topics found in CS1 syllabi analysis (Chapter 2).

Group	**MC^3 Categories**
Algorithm representations	N/A
Basic concepts of algorithm construction	A, B
Composite variables	C, E
Control structures	B, C, F
Functions, scope and parameter usage	D
Recursion	D

5.2 SO2: Development of valid artifacts that address MC^3

The goal of this objective was to develop artifacts that would directly and indirectly assist CS1 instructors and students in addressing MC^3 within the course. This objective yielded two key artifacts: an automated detection tool capable of identifying 14 of the 15 most severe MC^3 and a set of educational materials, comprised of short videos, lecture slides,

and flashcards. Validity from these artifacts stemmed from two facts: they were created using the insights and findings in SO1; and their elaboration followed methodologies or guidelines present in the literature.

Focusing on identifying the 15 most severe MC³ in SO1 was a deliberate research design choice to streamline the scope of this work. However, even within this focused scope, implementing the automated detection proved to be the most challenging aspect of this thesis. Analyzing the Abstract Syntax Tree is a common method for implementing static code analysis in similar research [39, 47, 59, 75], and authors have reported inherent challenges in this process. As detailed in Chapter 3, the developed tool for this thesis can effectively detect 14 out of the 15 most severe MC³. Nevertheless, several simplifications were necessary during implementation, primarily due to the anticipated low complexity of the analyzed code authored by CS1 students. For instance, simplifications included limiting the detection of MC³ C1 (`while` condition tested again inside its block) for only one `while` condition, since it would be rare for students to declare more. Another example was limiting detection of MC³ B8 (non utilization of `elif/else` statements) to only consider `if-elif` sequences without an `else` clause.

Another important aspect of the automated detection design was the incorporation of instructors' expertise. The necessity for this design became apparent when I had to address questions like "How many iterations of a `for`-loop would suggest that a `while` loop should have been used instead?" for MC³ C4 (arbitrary number of `for` loop executions instead of `while`). Similar questions arose when considering the maximum allowable number of declared lists that would indicate a misuse of this structure (MC³ E2 - redundant or unnecessary use of lists) and, perhaps the most challenging one, was determining the minimum number of characters required for a variable or function name to be considered significant (MC³ G4 - variables/functions with non significant names). Given the context-dependent nature of these MC³, instructors are best positioned to make such determinations, having developed each CS1 assignment and understanding what to expect from their students. The use of threshold constants to identify these MC³ is akin to the symptom-based approach discussed in Evans et al. [47], where the presence of a symptom may or may not indicate a misconception. The instructor's expertise is also required in EduLint[1], a Python linter designed for novice programmers. EduLint performs code quality checks using static code analysis, and some of the detected flaws are similar to MC³. However, since not all checks may be relevant for novices, EduLint's guidelines suggest that instructors configure which defects will trigger warnings.

The development of educational materials served as a complement to interventions addressing MC³ in CS1 courses. This initiative stemmed from personal interest and a lack of existing research (at least as far I was able to find) specifically targeting novice programmers' misconceptions [36, 97, 100]. Short videos and flashcards can also help students by providing spaced repetition [22, 43] in order to reactivate and reach peak retention of the contents taught. Previous research on the use of videos in CS1 provided guidelines for future content creation [129, 137], emphasizing appropriate length, language format, and cost-effectiveness attributes. It is crucial for instructors to ensure that future classes will benefit from these resources due to the potential production costs associated

[1] `https://edulint.readthedocs.io/en/latest/`

with video creation. In light of this, the developed educational materials in this thesis were shared with MC102 instructors. The large-scale analysis conducted in Chapter 4 highlighted the persistence of the most severe MC³ in CS1 classes, with nine appearing most frequently. This finding justified the creation of videos for use in future MC102 or similar courses, aligning with the cost-effectiveness importance mentioned. Moreover, since the guidelines generally suggests for short videos, the use of YouTube Shorts was considered due to its popularity among educational channels[2][3]. I had hoped this format would appeal to students' attention. Lastly, the proper language was achieved by using code examples created by CS1 students in the original assessment that led to the identification of MC³. By using these examples —and even reinforcing during lectures these examples were created by previous students— the attention impact would be enhanced.

Both short videos and flashcards were created to complement each other, guided by the same elaboration rationale outlined above. As I am not a designer, I encountered challenges assembling these materials, particularly due to a lack of proficiency in platforms like CapCut[4] or Canva[5]. However, the most daunting task was adapting content to fit within 60-second videos or a single-page digital flashcard. Consequently, I opted to use lecture slides to explain MC³ B8, E2, and G4, as I determined that these MC³ required more extensive explanations. The format was the only distinction between these slides and the other materials, which were all developed with the same methodology in mind.

5.3 SO3: Assessment of artifacts in a CS1 teaching environment

This objective aimed to evaluate the developed artifacts from SO2 within a CS1 class, conducted as a case study in an MC102 course with a total of 23 student participants. At the start of the course, students were divided into two groups, A and B. Throughout the course, each group received a set of short videos and flashcards addressing specific MC³ (refer to Table 4.5). The lecture slides were administered to all students, but only participants had their code analyzed. After the intervention period concluded, each group's MC³ occurrences were compared internally to analyze whether individual students who watched the corresponding educational materials showed changes in MC³ occurrences. The tool for automated detection assisted in this frequency analysis, although the tool itself was not directly assessed with respect to the students.

As discussed in Chapter 3, the distribution of materials was structured to continually introduce novel videos and flashcards to both groups, following the principles of spaced repetition. The timing of material dissemination during the CS1 course was determined by when each MC³ was observed to peak (based on Table 3.4 and Figures 4.2, 4.3, and 4.4). Similarly, the scheduling of lecture slides was aligned with these observations. However, as previously mentioned, I encountered challenges with this approach when I noticed

[2] https://www.youtube.com/@funwithpython8618/shorts
[3] https://www.youtube.com/@fastprogramming9916/shorts
[4] www.capcut.com
[5] www.canva.com

that students were not consistently viewing the YouTube-hosted videos. To address this wavering interest issue, I sent friendly reminders each week via email and during lectures to encourage engagement with the materials. Maintaining regular communication with participants proved essential in addressing this common issue with this kind of research.

Group A's results (Table 4.6) suggest that the materials had a positive impact on mitigating MC³ occurrences. This conclusion is supported by the comparison of total submissions with and without MC³ based on whether students had watched the materials. Specifically, the total submissions with MC³ were lower when students had watched the materials compared to when they had not. Additionally, the total submissions without any MC³ after watching the materials were higher than when they had not watched. In contrast, Group B (Table 4.7) yielded different results, with total submissions showing MC³ occurrences three times higher than when students had not watched the materials. This could indicate that the materials had a potentially negative impact on mitigating MC³ occurrences in Group B. However, the Chi-Squared test rejected the null hypothesis for Group A but not for Group B. In other words, the results from Group A suggest that the materials were beneficial, whereas the results for Group B were inconclusive.

What could explain the divergence between the two groups? Despite the nearly equal arrangement of students and consistent material distribution methods across both groups, attention is drawn to the nature of the MC³ themselves. It is possible that MC³ A4, C1, and D4 were more effectively explained and understood through short videos and flashcards compared to MC³ B9, C8, and G5. Additionally, although Table 4.5 suggests that Group B displayed slightly more interest in watching the materials than Group A, this did not translate into a positive outcome in the results. Furthermore, the fact that MC³ B8, D4, and G5 are among the most frequent (Figure 4.4) implies that they may be more challenging to address effectively.

Ultimately, my conclusions suggest that addressing MC³ through media formats like videos and flashcards is valuable but insufficient on its own. Their incorporation into CS1 courses must be coupled with a clear emphasis on the importance of avoiding MC³, and this emphasis must be reinforced periodically by instructors and teaching assistants during the CS1 course. This is especially crucial in courses where automatically graded assignments heavily influence students' final grades, as is the case with MC102. If instructors opt to penalize students' solutions with MC³, they should justify it by using code readability and maintainability attributes. This is because justifications on code total running time or resource consumption is not suited for CS1 students [70]. The goal of this thesis is heavily influenced by this statement, since its purpose is but to support instructors and teaching assistants to help students develop strong programming skills.

Lastly, another aspect of SO3 involved assessing the reflections on the adoption of MC³ interventions in CS1 classes. Although MC102 students provided feedback on the research they participated in, these responses were not included in the articles represented in Chapters 3 and 4 due to space constraints and a low response rate. However, I would like to share some of these responses here, as I have come to appreciate the value of context to understand situations better. The responses were originally in Brazilian Portuguese.

"*I would just like to thank the opportunity to participate in the research. In the beginning I had no idea that I would receive feedback in the way they were given and, undoubtedly, they were extremely valuable and made me realize many mistakes I was committing. They have also provided a sense of security in writing code with the presented commands.*"

(A male student that participated in the semi-structured observation).

"*I would like to thank the opportunity to participate in the project, it was extremely useful for me as I learned many things I had not seen in the classroom.*"

(A female student that participated in the semi-structured observation).

"*As a novice programmer, I consider that this research and its educational materials were of great assistance to prevent redundancies and imprecision. I would also like to emphasize that the materials addressing 'Pitfalls' were extremely important to help me avoid common flaws in my code development, as an excessive use of lists, confuse variable naming, etc.*"

(A male student from Group B that participated in the case study).

"*I did not perceive any change regarding elaborating my code as a programmer that would follow coding patterns, especially since this is not my goal with programming. However, I began to be more consistent in the way I have elaborated my code from one assignment to another.*"

(A male student from Group A that participated in the case study.)

Chapter 6

Reflections at Journey's End

"For this journey's end is but a step forward to tomorrow."
—"**Flow**". Music written by Masayoshi Soken.

This thesis aimed to enhance the teaching and learning of undergraduate introductory programming courses (CS1) by investigating coding behaviors exhibited by students that may indicate incomplete or incorrect comprehension of course topics. These behaviors, identified in functionally correct code, were termed Misconceptions in Correct Code (MC^3). All analysis primarily focused on the teaching context of the Algorithm and Computer Programming (MC102) course at Universidade Estadual de Campinas, which uses the Python programming language within the procedural paradigm. The thesis was presented as an article collection (Chapters 2 to 4), with each chapter representing a contribution to the study of MC^3.

Chapter 2 presented an analysis of CS1 course syllabi from Brazilian public universities. The primary objective was to assess the alignment of covered topics with those taught in MC102, thereby evaluating one replicability factor of this thesis' studies. A total of 225 syllabi from 95 public universities were analyzed, identifying 12 topics grouped into six categories: Algorithm representations; Basic concepts of algorithm construction; Composite variables; Control structures; Functions, scope, and parameter usage; and Recursion. The most recent version of the MC102 syllabus covers all these categories except for Algorithm representations. The analysis also revealed that C is the most commonly taught programming language (53%), followed by Python (8%), with the procedural paradigm being the predominant approach. These findings indicate diverse teaching contexts where MC^3 could manifest, but they are not enough to underscore their reasons for occurrence.

Chapter 3 detailed the methodology employed to identify and assess the list of MC^3. A total of 45 MC^3 were identified through manual assessment of 2,441 student submissions. Following consultations with 32 CS1 instructors, 15 MC^3 were classified as most severe, warranting deeper investigation. Instructors also shared insights into why students exhibit these coding behaviors and proposed teaching interventions to address them. The investigation into MC^3 occurrence also included semi-structured observations within MC102 classes, involving 20 students and one instructor. Results revealed that eight of the 15 most severe MC^3 are directly related to Boolean expressions and iteration, core concepts

in procedural-based programming. Evidence suggested that students often exhibit misconceptions about Python constructs such as decision statements and loops. Additionally, a careless coding approach, where functionality takes precedence over other aspects, was also identified as a contributing factor to MC^3 occurrence. The chapter also introduced an automated detection tool capable of identifying 14 out of the 15 most severe MC^3, which was used in assessing their frequencies throughout MC102 academic semesters.

Chapter 4 further explored MC^3 frequencies through a comprehensive study spanning eight academic terms of MC102. Over 40,000 student submissions were analyzed with the developed automated detection tool, revealing that MC^3 exhibit varying distributions. Analysis also revealed that the general frequency of MC^3 development is persistent, indicating that they do not disappear if no intervention is conducted. This insight inspired the creation of educational materials targeting nine of the most frequent MC^3. These materials, consisting of short videos, flashcards, and lecture slides, were designed to engage students and reinforce why these coding behaviors should be avoided. A case study involving 23 students assessed the impact of these materials, with results indicating a positive effect on mitigating some MC^3 in one group of students while yielding inconclusive results in another. These results elicit the importance of integrating such materials within the CS1 curriculum but, to increase their effectiveness, instructors and teaching assistants must continuously reinforce why MC^3 should be avoided. These materials should also be created with a proper language and a compelling format to increase students' engagement.

The thesis' concluding statement is that CS1 students can develop code that produces the expected outcome while still developing coding behaviors that elicits an incomplete or faulty comprehension of the CS1 topics. Although identified in Python, MC^3 can manifest in other imperative programming languages, suggesting that these behaviors are rooted in students' cognitive processes. Subsequently, as students' cognitive processes are highly influenced by their learning environment, this research emphasizes that to effectively address MC^3, code attributes such as readability and maintainability must be reinforced beyond correctness. The role of instructors and teaching assistants is crucial in mitigating the occurrence of MC^3, as they can unintentionally reinforce these behaviors through educational materials and teaching practices.

Future research on MC^3 could focus on developing additional materials for use during CS1 lectures. For example, creating a concept inventory tailored to address MC^3 behaviors could serve as an initial step toward implementing Active Learning techniques like Peer Instruction. While I did not have the opportunity to explore this idea, I believe it could be effectively applied in CS1 teaching environments, whether in classrooms or laboratories. Since students can interact with multiple-choice questionnaires using smartphones or clickers, lectures in classrooms can benefit from this approach without requiring computers.

Enhancing the automated detection of MC^3 while also providing formative feedback directly to students is another possibility of future research. A few years ago, I might have suggested that feedback should be solely provided by an autograder detecting MC^3. However, given the increasing interest in using Large Language Models (LLMs) in CS1 teaching, I now argue that leveraging this technology to address and provide feedback on MC^3 behaviors is a more promising approach. Recent research by Wang et al. [141]

explored different types of generated feedback, primarily focused on syntax errors. The authors reported that OpenAI's ChatGPT consistently generated more helpful messages compared to traditional Python error messages. Investigating whether similar feedback from LLMs can effectively address MC^3 would be valuable.

Lastly, I do not disregard the importance of exploring MC^3 occurrence in other CS1 teaching contexts, including demographic factors such as gender, previous programming experience, and type of higher education institution. Another important consideration is students' prior submissions before the final one containing MC^3. While this thesis demonstrated that assignment deadlines do not significantly impact MC^3 occurrence, it is possible that students who exhibit these behaviors were previously struggling to ensure their code passed all test cases. These attributes can contribute to creating more nuanced student profiles in Learning Analytics, for example. Although the presence of MC^3 alone may not determine a student's likelihood of failing the CS1 course, other related aspects could be inferred and studied further.

The reason behind this chapter's name is that I have come to realize the metaphor that "Ph.D. is journey". This realization coalesced because I once heard that "good research leaves the seed for other research" and that "a thesis must tell a story". While I do not believe this metaphor applies to research as an *entity*, it does apply to research *cycles*. As for my cycle, while I certainly do not consider myself as a hero, many were its similarities with the stages of the Hero's Journey, as listed by Vogler [139]. I experienced my call to adventure, faced refusals and challenges, sought guidance from mentors, confronted inner trials, and emerged with new insights akin to the hero's return with the elixir, created to improve the ordinary world of teaching and learning in CS1.

Bibliography

[1] Cristina L. Abad, Eduardo Ortiz-Holguin, and Edwin F. Boza. Have We Reached Consensus? An Analysis of Distributed Systems Syllabi. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 1082–1088, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380621. doi: 10.1145/3408877.3432409.

[2] Ashish Aggarwal, Christina Gardner-McCune, and David S. Touretzky. Evaluating the effect of using physical manipulatives to foster computational thinking in elementary school. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 9–14, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346986. doi: 10.1145/3017680.3017791. URL https://doi.org/10.1145/3017680.3017791.

[3] Murtaza Ali, Sourojit Ghosh, Prerna Rao, Raveena Dhegaskar, Sophia Jawort, Alix Medler, Mengqi Shi, and Sayamindu Dasgupta. Taking stock of concept inventories in computing education: A systematic literature review. ICER '23, page 397–415, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399760. doi: 10.1145/3568813.3600120. URL https://doi.org/10.1145/3568813.3600120.

[4] Vicki L. Almstrum, Peter B. Henderson, Valerie Harvey, Cinda Heeren, William Marion, Charles Riedesel, Leen-Kiat Soh, and Allison Elliott Tew. Concept inventories in computer science for the topic discrete mathematics. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '06, page 132–145, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595936033. doi: 10.1145/1189215.1189182. URL https://doi.org/10.1145/1189215.1189182.

[5] Amjad Altadmri and Neil C.C. Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, page 522–527, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450329668. doi: 10.1145/2676723.2677258. URL https://doi.org/10.1145/2676723.2677258.

[6] Ada Araujo, Daniel Filho, Elaine Oliveira, Leandro Carvalho, Filipe Pereira, and David Oliveira. Mapeamento e análise empírica de misconceptions comuns em

avaliações de introdução à programação. In *Anais do Simpósio Brasileiro de Educação em Computação*, pages 123–131, Porto Alegre, RS, Brasil, 2021. SBC. doi: 10.5753/educomp.2021.14478.

[7] Luis Araujo, Roberto Bittencourt, and Christina Chavez. Python Enhanced Error Feedback: Uma IDE Online de Apoio ao Processo de Ensino-Aprendizagem em Programação. In *Anais do Simpósio Brasileiro de Educação em Computação*, pages 326–333, Porto Alegre, RS, Brasil, 2021. SBC. doi: 10.5753/educomp.2021.14500.

[8] Richard H. Austing, Bruce H. Barnes, Della T. Bonnette, Gerald L. Engel, and Gordon Stokes. Curriculum '78: Recommendations for the Undergraduate Program in Computer Science— a Report of the ACM Curriculum Committee on Computer Science. *Commun. ACM*, 22(3):147–166, mar 1979. ISSN 0001-0782. doi: 10.1145/359080.359083.

[9] Nikolaos Avouris. Introduction to Computing: A Survey of Courses in Greek Higher Education Institutions. In *Proceedings of the 22nd Pan-Hellenic Conference on Informatics*, PCI '18, page 64–69, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450366106. doi: 10.1145/3291533.3291549.

[10] Elisa Baniassad, Lucas Zamprogno, Braxton Hall, and Reid Holmes. Stop the (autograder) insanity: Regression penalties to deter autograder overreliance. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 1062–1068, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380621. doi: 10.1145/3408877.3432430. URL https://doi.org/10.1145/3408877.3432430.

[11] Alexandre Barbosa, Evandro Costa, and Patrick Brito. Juízes online são suficientes ou precisamos de um var? In *Anais do III Simpósio Brasileiro de Educação em Computação*, pages 386–394, Porto Alegre, RS, Brasil, 2023. SBC. doi: 10.5753/educomp.2023.228224. URL https://sol.sbc.org.br/index.php/educomp/article/view/23909.

[12] Brett A. Becker. A Survey of Introductory Programming Courses in Ireland. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 58–64, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368957. doi: 10.1145/3304221.3319752.

[13] Brett A. Becker and Thomas Fitzpatrick. What do cs1 syllabi reveal about our expectations of introductory programming students? In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, page 1011–1017, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450358903. doi: 10.1145/3287324.3287485. URL https://doi.org/10.1145/3287324.3287485.

[14] Brett A. Becker and Keith Quille. 50 Years of CS1 at SIGCSE: A Review of the Evolution of Introductory Programming Education Research. In *Proceedings of the*

50th ACM Technical Symposium on Computer Science Education, SIGCSE '19, page 338–344, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450358903. doi: 10.1145/3287324.3287432.

[15] Brett A Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. Effective compiler error message enhancement for novice programming students. *Computer Science Education*, 26(2-3):148–175, 2016. doi: 10.1080/08993408.2016.1225464. URL https://doi.org/10.1080/08993408.2016.1225464.

[16] Brett A. Becker, Kyle Goslin, and Graham Glanville. The effects of enhanced compiler error messages on a syntax error debugging test. *SIGCSE 2018 - Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 2018-Janua: 640–645, 2018. doi: 10.1145/3159450.3159461.

[17] Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G. Feitelson. Meaningful identifier names: The case of single-letter variables. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 45–54, 2017. doi: 10.1109/ICPC.2017.18.

[18] Marc Berges and Peter Hubwieser. Concept Specification Maps: Displaying Content Structures. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, page 291–296, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320788. doi: 10.1145/2462476.2462503.

[19] Dave Binkley, Dawn Lawrie, Steve Maex, and Christopher Morrell. Impact of limited memory resources. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, page 83–92, USA, 2008. IEEE Computer Society. ISBN 9780769531762. doi: 10.1109/ICPC.2008.31. URL https://doi.org/10.1109/ICPC.2008.31.

[20] Glenn D Blank, Sally Hiestand, and Fang Wei. Overcoming misconceptions about computer science with multimedia. In *Proceedings of 35th SIGCSE Technical Symposium on Computer science Education*, 2004.

[21] Paulo Blikstein and Sepi Hejazi Moghadam. *Computing Education Literature Review and Voices from the Field*, page 56–78. Cambridge Handbooks in Psychology. Cambridge University Press, 2019.

[22] C.C. Bonwell and J.A. Eison. *Active Learning: Creating Excitement in the Classroom*. J-B ASHE Higher Education Report Series (AEHE). Wiley, 1991. ISBN 9781878380081.

[23] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. "i know it when i see it" perceptions of code quality: Iticse '17 working group report.

In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*, ITiCSE-WGR '17, page 70–85, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356275. doi: 10.1145/3174781.3174785. URL https://doi.org/10.1145/3174781.3174785.

[24] Yorah Bosse and Marco Gerosa. Reprovações e Trancamentos nas Disciplinas de Introdução à Programação da Universidade de São Paulo: Um Estudo Preliminar. In *Anais do XXIII Workshop sobre Educação em Computação*, pages 426–435, Porto Alegre, RS, Brasil, 2015. SBC. doi: 10.5753/wei.2015.10259.

[25] Carla E. Brodley, Benjamin J. Hescott, Jessica Biron, Ali Ressing, Melissa Peiken, Sarah Maravetz, and Alan Mislove. Broadening Participation in Computing via Ubiquitous Combined Majors (CS+X). In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2022, page 544–550, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450390705. doi: 10.1145/3478431.3499352.

[26] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. Blackbox: a large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, page 223–228, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326056. doi: 10.1145/2538862.2538924. URL https://doi.org/10.1145/2538862.2538924.

[27] William H Brown, Raphael C Malveau, Hays W" Skip" McCormick, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[28] Ricardo Caceffo, Steve Wolfman, Kellogg S. Booth, and Rodolfo Azevedo. Developing a Computer Science Concept Inventory for Introductory Programming. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, page 364–369, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450336857. doi: 10.1145/2839509.2844559.

[29] Ricardo Caceffo, Guilherme Gama, and Rodolfo Azevedo. Exploring Active Learning Approaches to Computer Science Classes. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, page 922–927, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450351034. doi: 10.1145/3159450.3159585.

[30] Ricardo Caceffo, Pablo Frank-Bolton, Renan Souza, and Rodolfo Azevedo. Identifying and validating java misconceptions toward a cs1 concept inventory. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 23–29, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368957. doi: 10.1145/3304221.3319771. URL https://doi.org/10.1145/3304221.3319771.

[31] Andrew Cain and Muhammad Ali Babar. Reflections on applying constructive alignment with formative feedback for teaching introductory programming and software architecture. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, page 336–345, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342056. doi: 10.1145/2889160.2889185. URL https://doi.org/10.1145/2889160.2889185.

[32] Tânia Alencar de Caldas. Desempenho dos alunos numa disciplina introdutória de programação e o conhecimento longevo. Master's thesis, Universidade Estadual de Campinas, 2020. URL https://hdl.handle.net/20.500.12733/1640661.

[33] RLBL Campos. Metodologia ERM2C: Para melhoria do processo de ensino-aprendizagem de lógica de programação. In *XVIII Workshop sobre Educação em Computação*, pages 961–970. XXX Congresso da Sociedade Brasileira de Computação, 2010. URL https://docplayer.com.br/52972769-Metodologia-erm2c-para-melhoria-do-processo-de-ensino-aprendizagem-de-logica-de-programacao.html.

[34] Simon Caton, Seán Russell, and Brett A. Becker. What fails once, fails again: Common repeated errors in introductory programming automated assessments. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1*, SIGCSE 2022, page 955–961, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450390705. doi: 10.1145/3478431.3499419. URL https://doi.org/10.1145/3478431.3499419.

[35] CC2020 Task Force. *Computing Curricula 2020: Paradigms for Global Computing Education*. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450390590. doi: https://doi.org/10.1145/3467967.

[36] Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L. Santos, and Matthias Hauswirth. A curated inventory of programming language misconceptions. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, page 380–386, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382144. doi: 10.1145/3430665.3456343. URL https://doi.org/10.1145/3430665.3456343.

[37] Louis Cohen, Lawrence Manion, and Keith Morrison. *Research Methods in Education*. Routledge, 2005.

[38] Catherine H. Crouch and Eric Mazur. Peer Instruction: Ten years of experience and results. *American Journal of Physics*, 69(9):970–977, 09 2001. ISSN 0002-9505. doi: 10.1119/1.1374249. URL https://doi.org/10.1119/1.1374249.

[39] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B. Rowe, and Nasser Giacaman. Understanding semantic style by analysing student code. In *Proceedings of the 20th Australasian Computing Education Conference*, ACE '18, page 73–82, New York, NY, USA, 2018. Association for Computing Machinery.

ISBN 9781450363402. doi: 10.1145/3160489.3160500. URL https://doi.org/10.1145/3160489.3160500.

[40] Zachary Dodds, Ran Libeskind-Hadas, and Eliot Bush. When CS 1 is Biology 1: Crossdisciplinary Collaboration as CS Context. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, page 219–223, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605588209. doi: 10.1145/1822090.1822152.

[41] Benedict du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986. doi: 10.2190/3LFX-9RRF-67T8-UVK9. URL https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9.

[42] Rodrigo Duran, Juha Sorva, and Otto Seppälä. Rules of program behavior. *ACM Trans. Comput. Educ.*, 21(4), nov 2021. doi: 10.1145/3469128. URL https://doi.org/10.1145/3469128.

[43] Hermann Ebbinghaus. Memory: a contribution to experimental psychology. *Annals of neurosciences*, 20(4):155—156, October 2013. ISSN 0972-7531. doi: 10.5214/ans.0972.7531.200408. URL https://europepmc.org/articles/PMC4117135.

[44] Mary B. Eberly, Sarah E. Newton, and Robert A. Wiggins. THE SYLLABUS AS A TOOL FOR STUDENT-CENTERED LEARNING. *The Journal of General Education*, 50(1):56–74, 2001. ISSN 00213667, 15272060. URL http://www.jstor.org/stable/27797862.

[45] Stephen H. Edwards. Automated feedback, the next generation: Designing learning experiences. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 610–611, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380621. doi: 10.1145/3408877.3437225. URL https://doi.org/10.1145/3408877.3437225.

[46] Margaret Ellis, Clifford A. Shaffer, and Stephen H. Edwards. Approaches for coordinating etextbooks, online programming practice, automated grading, and more into one course. *SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 126–132, 2019. doi: 10.1145/3287324.3287487.

[47] Abigail Evans, Zihan Wang, Jieren Liu, and Mingming Zheng. Side-lib: A library for detecting symptoms of python programming misconceptions. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, page 159–165, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701382. doi: 10.1145/3587102.3588838. URL https://doi.org/10.1145/3587102.3588838.

[48] Rafael Corsi Ferrao, Igor dos Santos Montagner, Mariana Silva, Craig Zilles, and Rodolfo Azevedo. Embedded-check: A code quality tool for automatic firmware verification. In *ITiCSE 2024*, Association for Computing Machinery. Association for Computing Machinery, 2024. In Press.

[49] FIA. Universidades públicas: O que são, importância e lista de instituições, 2019. URL https://fia.com.br/blog/universidades-publicas/. Online.

[50] Björn Fischer, Fabian Birk, Eva-Maria Iwer, Sven Eric Panitz, and Ralf Dörner. Addressing misconceptions in introductory programming: Automated feedback in integrated development environments. In *Proceedings of the 15th International Conference on Education Technology and Computers*, ICETC '23, page 1–8, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400709111. doi: 10.1145/3629296.3629297. URL https://doi.org/10.1145/3629296.3629297.

[51] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. Modernizing plan-composition studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, page 211–216, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450336857. doi: 10.1145/2839509.2844556. URL https://doi.org/10.1145/2839509.2844556.

[52] Nadjim Fréchet, Justin Savoie, and Yannick Dufresne. Analysis of Text-Analysis Syllabi: Building a Text-Analysis Syllabus Using Scaling. *PS: Political Science & Politics*, 53(2):338–343, 2020. doi: 10.1017/S1049096519001732.

[53] Leandro Galvão, David Fernandes, and Bruno Gadelha. Juiz online como ferramenta de apoio a uma metodologia de ensino híbrido em programação. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação - SBIE)*, 27(1):140, 2016. ISSN 2316-6533. doi: 10.5753/cbie.sbie.2016.140. URL http://milanesa.ime.usp.br/rbie/index.php/sbie/article/view/6694.

[54] Guilherme Gama, Ricardo Caceffo, Renan Souza, Raysa Bennati, Tales Aparecida, Islene Garcia, and Rodolfo Azevedo. An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in Python. Technical Report IC-18-19, Institute of Computing, University of Campinas, November 2018.

[55] Gabriel Gomes, Ygor Nishi, Paula Ramos, Leonardo Silva, Eduardo Guerra, Igor Wiese, and Yorah Bosse. Um recurso educacional para desenvolver a habilidade da percepção de padrões de equívocos com aprendizes de programação. In *Anais do III Simpósio Brasileiro de Educação em Computação*, pages 328–336, Porto Alegre, RS, Brasil, 2023. SBC. doi: 10.5753/educomp.2023.228330. URL https://sol.sbc.org.br/index.php/educomp/article/view/23903.

[56] Remo Gresta, Vinicius Durelli, and Elder Cirilo. Naming practices in object-oriented programming: An empirical study. *Journal of Software Engineering Research and Development*, 11(1):5:1 – 5:16, Feb. 2023. doi: 10.5753/jserd.2023.2582. URL https://journals-sol.sbc.org.br/index.php/jserd/article/view/2582.

[57] Judith Grunert. *The Course Syllabus: A Learning-Centered Approach*. Anker Publishing Co. Inc., Bolton, MA, USA, 1 edition, 1997. ISBN 9781882982189.

[58] Philip Guo. Python Is Now the Most Popular Introductory Teaching Language at Top U.S. Universities, 2014. URL https://cacm.acm.org/blogs/blog-

cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext. Online.

[59] Luke Gusukuma, Austin Cory Bart, and Dennis Kafura. Pedal: An infrastructure for automated feedback systems. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 1061–1067, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367936. doi: 10.1145/3328778.3366913. URL https://doi.org/10.1145/3328778.3366913.

[60] Mark Guzdial and Benedict du Boulay. *The History of Computing Education Research*, page 11–39. Cambridge Handbooks in Psychology. Cambridge University Press, 2019.

[61] Sally Hamouda, Stephen H Edwards, Hicham G Elmongui, Jeremy V Ernst, and Clifford A Shaffer. A basic recursion concept inventory. *Computer Science Education*, 27(2):121–148, 2017. doi: 10.1080/08993408.2017.1414728. URL https://doi.org/10.1080/08993408.2017.1414728.

[62] Matthew Hertz. What Do "CS1" and "CS2" Mean? Investigating Differences in the Early Courses. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, page 199–203, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300063. doi: 10.1145/1734263.1734335.

[63] Matthew Hertz and Sarah Michele Ford. Investigating Factors of Student Learning in Introductory Courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 195–200, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318686. doi: 10.1145/2445196.2445254.

[64] Gregory W. Hislop, Lillian Cassel, Lois Delcambre, Edward Fox, Rick Furuta, and Peter Brusilovsky. Ensemble: Creating a National Digital Library for Computing Education. In *Proceedings of the 10th ACM Conference on SIG-Information Technology Education*, SIGITE '09, page 200, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587653. doi: 10.1145/1631728.1631783.

[65] Jack Hollingsworth. Automatic graders for programming classes. *Commun. ACM*, 3(10):528–529, October 1960. ISSN 0001-0782. doi: 10.1145/367415.367422. URL https://doi.org/10.1145/367415.367422.

[66] Silas Hsu, Tiffany Wenting Li, Zhilin Zhang, Max Fowler, Craig Zilles, and Karrie Karahalios. Attitudes surrounding an imperfect ai autograder. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380966. doi: 10.1145/3411764.3445424. URL https://doi.org/10.1145/3411764.3445424.

[67] Petri Ihantola and Andrew Petersen. Code complexity in introductory programming courses. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*, pages 7662–7670, 2019.

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

[69] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and Renske Weeda. Fostering program comprehension in novice programmers - learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '19, page 27–52, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450375672. doi: 10.1145/3344429.3372501. URL https://doi.org/10.1145/3344429.3372501.

[70] Saj-Nicole A. Joni and Elliot Soloway. But My Program Runs! Discourse Rules for Novice Programmers. *Journal of Educational Computing Research*, 2(1):95–125, 1986. doi: 10.2190/6E5W-AR7C-NX76-HUT2.

[71] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, page 107–111, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300063. doi: 10.1145/1734263.1734299. URL https://doi.org/10.1145/1734263.1734299.

[72] Cazembe Kennedy and Eileen T. Kraemer. What are they thinking? eliciting student reasoning about troublesome concepts in introductory computer science. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, Koli Calling '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365352. doi: 10.1145/3279720.3279728. URL https://doi.org/10.1145/3279720.3279728.

[73] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, page 110–115, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450347044. doi: 10.1145/3059009.3059061. URL https://doi.org/10.1145/3059009.3059061.

[74] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. How teachers would help students to improve their code. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 119–125, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368957. doi: 10.1145/3304221.3319780. URL https://doi.org/10.1145/3304221.3319780.

[75] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. A tutoring system to learn code refactoring. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 562–568, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380621. doi: 10.1145/3408877.3432526. URL https://doi.org/10.1145/3408877.3432526.

[76] Päivi Kinnunen and Lauri Malmi. Why Students Drop out CS1 Course? In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, page 97–108, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934944. doi: 10.1145/1151588.1151604.

[77] Thomas Kluvyer. Green Tree Snakes - the missing Python AST docs. Retrieved on 06/05/2023 from link.

[78] Antti Laaksonen. *Guide to competitive programming*. Springer, 2020.

[79] Thomas Lancaster, Anthony V. Robins, and Sally A. Fincher. *Assessment and Plagiarism*, page 414–444. Cambridge Handbooks in Psychology. Cambridge University Press, 2019.

[80] N. Lasry. Clickers or Flashcards: Is There Really a Difference? *The Physics Teacher*, 46(4):242–244, 04 2008. ISSN 0031-921X. doi: 10.1119/1.2895678. URL https://doi.org/10.1119/1.2895678.

[81] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. *Research Methods in Human-Computer Interaction*. Morgan Kaufmann, 2017.

[82] Marcos A. P. Lima, Leandro S. G. Carvalho, Elaine H. T. de Oliveira, David B. F. de Oliveira, and Filipe D. Pereira. Uso de atributos de código para classificar a dificuldade de questões de programação em juízes online. *Revista Brasileira de Informática na Educação*, 29:1137–1157, set. 2021. doi: 10.5753/rbie.2021.29.0.1137.

[83] David Liu and Andrew Petersen. Static analyses in python programming courses. *SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 666–671, 2019. doi: 10.1145/3287324.3287503.

[84] Andrew Luxton-Reilly. Learning to Program is Easy. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, page 284–289, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342315. doi: 10.1145/2899415.2899432.

[85] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. Introductory Programming: A Systematic Literature Review. ITiCSE 2018 Companion, page 55–106, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450362238. doi: 10.1145/3293881.3295779.

[86] Lauri Malmi, Ian Utting, and Amy J. Ko. *Tools and Environments*, page 639–662. Cambridge Handbooks in Psychology. Cambridge University Press, 2019.

[87] Eric D. Manley and Timothy M. Urness. Video-based instruction for introductory computer programming. *J. Comput. Sci. Coll.*, 29(5):221–227, may 2014. ISSN 1937-4771.

[88] Samiha Marwan, Nicholas Lytle, Joseph Jay Williams, and Thomas Price. The impact of adding textual explanations to next-step hints in a novice programming environment. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 520–526, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368957. doi: 10.1145/3304221.3319759. URL https://doi.org/10.1145/3304221.3319759.

[89] Wilbert James McKeachie. *Teaching tips: A guidebook for the beginning college teacher*. D.C. Heath and Company, Lexington, MA, USA., 7 edition, 1978. ISBN 978066901151.

[90] Dawn McKinney and Leo F Denton. Houston, we have a problem: there's a leak in the cs1 affective oxygen tank. *ACM SIGCSE Bulletin*, 36(1):236–239, 2004.

[91] Priscilla Nascimento. Recomendação de ação pedagógica no ensino de introdução à programação por meio de raciocínio baseado em casos. Master's thesis, Programa de Pós-graduação em Informática, 2018. URL https://tede.ufam.edu.br/handle/tede/6837.

[92] Eduardo Oliveira, Hieke Keuning, and Johan Jeuring. Student code refactoring misconceptions. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, page 19–25, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701382. doi: 10.1145/3587102.3588840. URL https://doi.org/10.1145/3587102.3588840.

[93] Filipe Pereira, Samuel Fonseca, Elaine Oliveira, David Oliveira, Alexandra Cristea, and Leandro Carvalho. Deep learning for early performance prediction of introductory programming students: a comparative and explanatory study. *Revista Brasileira de Informática na Educação*, 28(0):723–748, 2020. ISSN 2317-6121. doi: 10.5753/rbie.2020.28.0.723.

[94] Filipe D. Pereira, Elaine H. T. Oliveira, David B. F. Oliveira, Alexandra I. Cristea, Leandro S. G. Carvalho, Samuel C. Fonseca, Armando Toda, and Seiji Isotani. Using learning analytics in the amazonas: understanding students' behaviour in introductory programming. *British Journal of Educational Technology*, 51(4): 955–972, 2020. doi: https://doi.org/10.1111/bjet.12953. URL https://bera-journals.onlinelibrary.wiley.com/doi/abs/10.1111/bjet.12953.

[95] Roberto Pereira, Leticia Peres, and Fabiano Silva. Hello World: 17 habilidades para exercitar desde o início da graduação em computação. In *Anais do Simpósio*

Brasileiro de Educação em Computação, pages 193–203, Porto Alegre, RS, Brasil, 2021. SBC. doi: 10.5753/educomp.2021.14485.

[96] Giovanni Kuckartz Pergher and Lilian Milnitsky Stein. Compreendendo o esquecimento: teorias clássicas e seus fundamentos experimentais. *Psicologia USP*, 14(1): 129–155, 2003. ISSN 0103-6564. doi: 10.1590/S0103-65642003000100008. URL https://doi.org/10.1590/S0103-65642003000100008.

[97] Rifca M Peters. Identifying and addressing common programming misconceptions with variables-part ii. Master's thesis, University of Twente, 2018.

[98] Andrew Petersen, Michelle Craig, Jennifer Campbell, and Anya Tafliovich. Revisiting Why Students Drop CS1. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, Koli Calling '16, page 71–80, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450347709. doi: 10.1145/2999541.2999552.

[99] Raymond S. Pettit, John Homer, and Roger Gee. Do enhanced compiler error messages help students? results inconclusive. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 465–470, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346986. doi: 10.1145/3017680.3017768. URL https://doi.org/10.1145/3017680.3017768.

[100] Danny Plass-Oude Bos. Identifying and addressing common programming misconceptions with variables (part 1). Master's thesis, University of Twente, 2015.

[101] Andres Jessé Porfirio, Roberto Pereira, and Eleandro Maschio. A-Learn EvId: A Method for Identifying Evidence of Computer Programming Skills Through Automatic Source Code Assessment. *Revista Brasileira de Informática na Educação*, 29: 692–717, jul. 2021. doi: 10.5753/rbie.2021.29.0.692.

[102] Leo Porter, Cynthia Bailey Lee, and Beth Simon. Halving fail rates using peer instruction: A study of four computer science courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 177–182, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318686. doi: 10.1145/2445196.2445250. URL https://doi.org/10.1145/2445196.2445250.

[103] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. Metacognitive difficulties faced by novice programmers in automated assessment tools. *ICER 2018 - Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 41–50, 2018. doi: 10.1145/3230977.3230981.

[104] David Pritchard. Frequency distribution of error messages. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*,

PLATEAU 2015, page 1–8, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450339070. doi: 10.1145/2846680.2846681. URL https://doi.org/10.1145/2846680.2846681.

[105] David Pritchard and Troy Vasiga. Cs circles: an in-browser python course for beginners. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 591–596, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318686. doi: 10.1145/2445196.2445370. URL https://doi.org/10.1145/2445196.2445370.

[106] Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE).*, 18(1), oct 2017. doi: 10.1145/3077618. URL https://doi.org/10.1145/3077618.

[107] Anthony V. Robins. *Novice Programmers and Introductory Programming*, page 327–376. Cambridge Handbooks in Psychology. Cambridge University Press, 2019.

[108] Anthony V. Robins. Dual process theories: Computing cognition in context. *ACM Trans. Comput. Educ.*, 22(4), sep 2022. doi: 10.1145/3487055.

[109] Inga Saatz and Andrea Kienle. Learning with e-flashcards – does it matter? In Davinia Hernández-Leo, Tobias Ley, Ralf Klamma, and Andreas Harrer, editors, *Scaling up Learning for Sustained Impact*, pages 629–630, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40814-4.

[110] Christopher Scaffidi, Mary Shaw, and Brad Myers. Estimating the Numbers of End Users and End User Programmers. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '05, page 207–214, USA, 2005. IEEE Computer Society. ISBN 0769524435. doi: 10.1109/VLHCC.2005.34.

[111] Carsten Schulte and Jens Bennedsen. What Do Teachers Teach in Introductory Programming? In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, page 17–28, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934944. doi: 10.1145/1151588.1151593.

[112] Robert M. Siegfried, Katherine G. Herbert-Berger, Kees Leune, and Jason P. Siegfried. Trends Of Commonly Used Programming Languages in CS1 And CS2 Learning. In *2021 16th International Conference on Computer Science & Education (ICCSE)*, pages 407–412, 2021. doi: 10.1109/ICCSE51940.2021.9569444.

[113] Diogo Correia Araujo Silva. Flashcards digitais - técnica de repetição espaçada aplicada ao apoio na memorização do conteúdo estudado. *Revista Gestão Universitária*, 1:1–10, 2015.

[114] Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. Análise estática de código em conjunto com autograders. In *Anais Estendidos do I Simpósio Brasileiro de Educação em Computação*, pages 25–26, Porto Alegre, RS, Brasil, 2021. SBC. doi:

10.5753/educomp_estendido.2021.14858. URL https://sol.sbc.org.br/index.php/educomp_estendido/article/view/14858.

[115] Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. Análise dos Tópicos Mais Abordados em Disciplinas de Introdução à Programação em Universidades Federais Brasileiras. In *Anais do II Simpósio Brasileiro de Educação em Computação*, pages 29–39, Porto Alegre, RS, Brasil, 2022. SBC. doi: 10.5753/educomp.2022.19196.

[116] Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. Misconceptions in Correct Code: rating the severity of undesirable programming behaviors in Python CS1 courses. Technical Report IC-23-01, Institute of Computing, University of Campinas, 2023.

[117] Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. Passar nos casos de teste é suficiente? identificação e análise de problemas de compreensão em códigos corretos. In *Anais do III Simpósio Brasileiro de Educação em Computação*, pages 119–129, Porto Alegre, RS, Brasil, 2023. SBC. doi: 10.5753/educomp.2023.228346. URL https://sol.sbc.org.br/index.php/educomp/article/view/23881.

[118] Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. A syllabi analysis of cs1 courses from brazilian public universities. *Brazilian Journal of Computers in Education*, 31 (1):407–436, Aug. 2023. doi: 10.5753/rbie.2023.2870. URL https://sol.sbc.org.br/journals/index.php/rbie/article/view/2870.

[119] Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. When Test Cases Are Not Enough: Identification, Assessment, and Rationale of Misconceptions in Correct Code (MC^3). *Brazilian Journal of Computers in Education*, 31:1165–1199, Dec. 2023. doi: 10.5753/rbie.2023.3552. URL https://sol.sbc.org.br/journals/index.php/rbie/article/view/3552.

[120] Beth Simon, Michael Kohanfars, Jeff Lee, Karen Tamayo, and Quintin Cutts. Experience report: Peer instruction in introductory computing. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, page 341–345, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300063. doi: 10.1145/1734263.1734381. URL https://doi.org/10.1145/1734263.1734381.

[121] Beth Simon, Sarah Esper, Leo Porter, and Quintin Cutts. Student experience in a student-centered peer instruction classroom. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, page 129–136, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322430. doi: 10.1145/2493394.2493407. URL https://doi.org/10.1145/2493394.2493407.

[122] Beth Simon, Julian Parris, and Jaime Spacco. How we teach impacts student learning: Peer instruction vs. lecture in cs0. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 41–46, New York,

NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318686. doi: 10.1145/2445196.2445215. URL https://doi.org/10.1145/2445196.2445215.

[123] Robert H. Sloan, Cynthia Taylor, and Richard Warner. Initial Experiences with a CS + Law Introduction to Computer Science (CS 1). In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, page 40–45, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450347044. doi: 10.1145/3059009.3059029.

[124] Robert H. Sloan, Valerie Barr, Heather Bort, Mark Guzdial, Ran Libeskind-Hadas, and Richard Warner. *CS + X Meets CS 1: Strongly Themed Intro Courses*, page 960–961. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450367936. doi: 10.1145/3328778.3366975.

[125] S. Sobral. 30 YEARS OF CS1: PROGRAMMING LANGUAGES EVOLUTION. In *ICERI2019 Proceedings*, 12th annual International Conference of Education, Research and Innovation, pages 9197–9205. IATED, 2019. ISBN 978-84-09-14755-7. doi: 10.21125/iceri.2019.2214.

[126] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1984. doi: 10.1109/TSE.1984.5010283.

[127] Juha Sorva and Arto Vihavainen. Break statement considered. *ACM Inroads*, 7(3): 36–41, aug 2016. ISSN 2153-2184. doi: 10.1145/2950065. URL https://doi.org/10.1145/2950065.

[128] Juha Sorva, Jan Lönnberg, and Lauri Malmi. Students' ways of experiencing visual program simulation. *Computer Science Education*, 23(3):207–238, 2013. doi: 10.1080/08993408.2013.807962. URL https://doi.org/10.1080/08993408.2013.807962.

[129] Ben Stephenson. Coding demonstration videos for cs1. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, page 105–111, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450358903. doi: 10.1145/3287324.3287445. URL https://doi.org/10.1145/3287324.3287445.

[130] Allison Elliott Tew and Mark Guzdial. Developing a Validated Assessment of Fundamental CS1 Concepts. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, page 97–101, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300063. doi: 10.1145/1734263.1734297.

[131] Allison Elliott Tew and Mark Guzdial. The FCS1: A Language Independent Assessment of CS1 Knowledge. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, page 111–116, New York,

NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305006. doi: 10.1145/1953163.1953200.

[132] Christine D Tippett. Refutation text in science education: A review of two decades of research. *International journal of science and mathematics education*, 8:951–970, 2010.

[133] Manas Tungare, Xiaoyan Yu, William Cameron, GuoFang Teng, Manuel A. Pérez-Quiñones, Lillian Cassel, Weiguo Fan, and Edward A. Fox. Towards a Syllabus Repository for Computer Science Courses. *SIGCSE Bull.*, 39(1):55–59, mar 2007. ISSN 0097-8418. doi: 10.1145/1227504.1227331.

[134] Leo C. Ureel II and Charles Wallace. Automated critique of early programming antipatterns. *SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 738–744, 2019. doi: 10.1145/3287324.3287463.

[135] Adilson Vahldick, Maria José Marcelino, and António José Mendes. Analyzing novices' fun and programming behaviours while playing a serious blocks-based game. *Revista Brasileira de Informática na Educação*, 29:1337–1355, dez. 2021. doi: 10.5753/rbie.2021.2069.

[136] David W. Valentine. CS Educational Research: A Meta-Analysis of SIGCSE Technical Symposium Proceedings. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, page 255–259, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581137982. doi: 10.1145/971300.971391.

[137] Hans Van der Meij and Jan Van der Meij. Eight guidelines for the design of instructional videos for software training. *Technical communication*, 60(3):205–228, 2013.

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

[139] Christopher Vogler. A practical guide to joseph campbell's the hero with a thousand faces. *Hero's Journey*, 1985.

[140] Henry M. Walker. ACM RETENTION COMMITTEE Retention of Students in Introductory Computing Courses: Curricular Issues and Approaches. *ACM Inroads*, 8(4):14–16, oct 2017. ISSN 2153-2184. doi: 10.1145/3151936.

[141] Sierra Wang, John Mitchell, and Chris Piech. A large scale rct on effective error messages in cs1. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2024, page 1395–1401, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704239. doi: 10.1145/3626252.3630764. URL `https://doi.org/10.1145/3626252.3630764`.

[142] Brian A Wichmann, AA Canning, DL Clutterbuck, LA Winsborrow, NJ Ward, and D William R Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, 10:69–75(6), March 1995.

[143] Allan Wigfield and Jacquelynne S. Eccles. Expectancy–Value Theory of Achievement Motivation. *Contemporary Educational Psychology*, 25(1):68–81, 2000. ISSN 0361-476X. doi: https://doi.org/10.1006/ceps.1999.1015.

[144] Wikipedia. Lista de universidades federais do brasil — wikipédia, a enciclopédia livre, 2021. URL `https://pt.wikipedia.org/w/index.php?title=Lista_de_universidades_federais_do_Brasil&oldid=61551015`. Online.

[145] Wikipedia. Lista de universidades estaduais do brasil — wikipédia, a enciclopédia livre, 2022. URL `https://pt.wikipedia.org/w/index.php?title=Lista_de_universidades_estaduais_do_Brasil&oldid=63117788`. Online.

[146] Wikipedia. Lista de universidades municipais do brasil — wikipédia, a enciclopédia livre, 2022. URL `https://pt.wikipedia.org/w/index.php?title=Lista_de_universidades_municipais_do_Brasil&oldid=63408268`. Online.

[147] Jeannette M. Wing. Computational Thinking. *Commun. ACM*, 49(3):33–35, mar 2006. ISSN 0001-0782. doi: 10.1145/1118178.1118215.

[148] Robert Zajonc. Mere Exposure: A Gateway to the Subliminal. *Current Directions in Psychological Science*, 10(6):224–228, 2001. doi: 10.1111/1467-8721.00154.

[149] Avelino Francisco Zorzo, Daltro Nunes, Ecivaldo Matos, Igor Steinmacher, Renata Mendes de Araujo, Ronaldo Correia, and Simone Martins. *Referenciais de Formação para os Cursos de Graduação em Computação*. Sociedade Brasileira de Computação (SBC), 2017. ISBN: 978-85-7669-424-3.

[150] Inez Zung, Megan N Imundo, and Steven C Pan. How do college students use digital flashcards during self-regulated learning? *Memory*, 30(8):923–941, 2022. doi: 10.1080/09658211.2022.2058553.

Appendix A

Catalog of Misconceptions in Correct Code

This appendix presents a catalog compiling all obtained information regarding the 45 MC^3 identified in this work. The catalog is divided into two sections: the first contains information on the 15 most severe MC^3, which are the focus of this thesis; while the second presents information on the other 30 identified MC^3. Both sections are listed by MC^3 IDs in lexicographical order, as represented in Table 5.1. Each MC^3 is described using the following attributes:

- **ID** and **Name**.
- **Description**: A brief explanation of how the MC^3 manifests in the Python code.
- **Example**: A simple example of Python code that contains the MC^3.
- **DIF**: An integer indicating the classified severity of the MC^3. It was calculated as the difference between instructors who agreed the MC^3 was severe and those who did not agree (refer to Table 3.3). The value of DIF ranges between [-22, 30], indicating that greater values represent greater severity.
- **Rationale**: An explanation of why students develop this MC^3. This explanation is a conclusion given the opinions gathered from CS1 instructors and students.
- **Consequences**: Describes possible future coding behaviors that can arise if the MC^3 is not addressed.
- **Intervention**: Describes how students can be guided during the CS1 course to mitigate the MC^3.

A.1 Most Severe MC^3

This section presents information on the 15 MC^3 that were classified as most severe in this thesis. The DIF values were used as a threshold to determine the most severe MC^3. In this case, all MC^3 with a DIF strictly greater than 10 were classified as most severe.

Category A: Variables, identifiers, and scope

A.1.1 A4: Redefinition of built-in

Description This MC³ occurs when a student mistakenly names a variable or function with the same name as a Python built-in function.

Example In Code A.1, the Python built-in `min()` is reassigned as a variable in line 7.

```
1 item1 = float(input("Value 1: "))
2 item2 = float(input("Value 2: "))
3 item3 = float(input("Value 3: "))
4
5 total = item1 + item2 + item3
6
7 min = 0.15 * total
8 print("Minimum value:", min)
```

Code A.1: A4: Redefinition of built-in.

DIF 12

Rationale This MC³ arises because students are not familiar with all Python built-ins initially. The choice of natural language for naming variables or functions (e.g., English, Spanish, or Portuguese) also affects the likelihood of this MC³. Overall, this MC³ can be classified as an unintentional, careless approach to coding.

Consequences This MC³ could lead to future errors that are difficult to identify if the student later tries to use the original built-in as intended.

Intervention Students should be advised that redefining a built-in should be avoided. Instructors and teaching assistants can point out how different programming environments distinguish built-ins from other declared names, such as by using different font colors.

Category B: Boolean expressions

A.1.2 B6: Boolean comparison attempted with `while` loop

Description Indicates an occurrence when students attempt to create a basic comparison statement using a `while` loop instead of appropriate conditional commands.

Example In Code A.2, the student declared a `while` loop instead of an `if` statement to check the condition regarding `num1` and `num2` in line 4.

```
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
```

Code A.2: B6: Boolean comparison attempted with `while` loop.

DIF 20

Rationale This MC³ arises because students might have difficulties distinguishing between general conditional statements and those used in loops. Another possible reason for this MC³ is the uneducated programming knowledge students may have acquired previously.

Consequences This MC³ indicates difficulties in understanding the concepts of conditional statements and loops. If left unaddressed, it might lead to future struggles with later topics.

Intervention Students should be instructed that loops are declared only when a block of code is expected to execute more than once. Instructors can highlight the difference between general conditional statements and `while` conditional statements through examples or exercises in lectures.

A.1.3 B8: Non utilization of `elif/else` statement

Description This MC³ occurs when students declare a sequence of `if-elif` statements without including a concluding `else` clause.

Example In Code A.3, the `elif` declared in line 6 could have been an `else`. Moreover, the `if` sequences declared in lines 9 and 11 can lead to overwriting the value of `res` depending on the value of `num2`.

```
1  num1 = int(input())
2  num2 = int(input())
3
4  if num1 % 2 == 0:
5      print(num1, "even")
6  elif num1 % 2 == 1:
7      print(num1, "odd")
8
9  if num2 <= 0:
10     res = num1 * num2
11 if num2 % 2 == 0:
12     res = num1 ** num2
```

Code A.3: B8: Non utilization of `elif/else` statement.

DIF 16

Rationale This MC³ arises because students might have difficulties understanding the differences between `if`, `elif`, and `else`. Specifically, students are struggling in identifying mutually exclusive conditions to be checked.

Consequences While substituting an `elif` with an `else` is generally due to a lack of attention, sequences of `if` statements with non-mutually exclusive conditions can result in unexpected code outputs.

Intervention Students should be instructed about the differences between `if`, `elif`, and `else`. Lecture examples can help in determining whether conditions are mutually exclusive. Instructors can also provide examples of unexpected code outputs resulting from a careless handling of non-mutually exclusive conditions.

A.1.4 B9: `elif/else` retesting already checked conditions

Description This MC³ occurs when students declare an `elif` statement checking the opposite condition of a previously declared `if` or `elif` statement.

Example In Code A.4, the `elif` declared in line 8 is retesting the opposite condition of an already verified `if` in line 5.

```
1  e1 = float(input("Edge 1: "))
2  e2 = float(input("Edge 2: "))
3  e3 = float(input("Edge 3: "))
4
5  if e1 == e2 == e3:
6      print("Scalene triangle")
7
8  elif (e1 == e2 or e1 == e3 or e2 == e3) and not e1 == e2 == e3:
9      print("Isosceles triangle")
10
11 else:
12     print("Scalene triangle")
```

Code A.4: B9: `elif/else` retesting already checked conditions.

DIF 14

Rationale This MC³ arises because students might have difficulties understanding that `elif` means "`else if`". Another possible scenario is that students are performing redundant checks to ensure the autograder does not incorrectly interpret their results.

Consequences Misconceptions about `elif` or misunderstandings about how the autograder works can lead to redundant complexity in the students' future code.

Intervention Students should be instructed about the differences between `if`, `elif`, and `else`. Instructors should also reinforce how the autograder system works during lectures.

A.1.5 B12: Consecutive equal `if` statements with distinct operations in their blocks

Description This MC³ occurs when students declare two or more `if` statements that test the same condition, albeit executing different commands in each block.

Example In Code A.5, the `if` statements declared in lines 4 and 6 check the same condition. They could have been merged in only one test.

```
1 num1 = int(input())
2 num2 = int(input())
3
4 if num1 == num2 * 2:
5     print(num1, "multiple␣of", num2)
6 if num1 == num2 * 2:
7     print(num1, "odd")
```

Code A.5: B12: Consecutive equal `if` statements with distinct operations in their blocks.

DIF 14

Rationale An underlying reason for this MC³ is that students explicitly want to separate two or more `if` statements in their code, possibly due to their reasoning in solving the assignment. It is also possible that students have the misconception that an `if` block can only encapsulate a limited number of declarations.

Consequences This MC³ can lead to redundant complexity in the students' future code.

Intervention Students should be instructed that `if`, `elif`, or `else` blocks can encapsulate multiple statements to avoid this type of redundancy.

Category C: Iteration

A.1.6 C1: `while` condition tested again inside its block

Description This MC³ occurs when students declare a `while` loop and, at the end of its body, also include an `if` statement to verify if the `while` condition has become false, ending the loop with a `break` statement.

Example In Code A.6, the condition of the `while` declared in line 4 is unnecessarily tested again in line 9.

```
1  sum_nums = total_nums = 0
2
3  num = int(input())
4  while num != 0:
5      sum_nums += num
6      total_nums += 1
7
8      num = int(input())
9      if num == 0:
10         break
11
12 average = sum_nums / total_nums
13 print("Avarage:", average)
```

Code A.6: C1: `while` condition tested again inside its block.

DIF 20

Rationale This MC³ arises because students might have difficulties understanding the concepts of the `while` command. However, the redundant check can also be deliberately done by students who want to ensure the autograder does not incorrectly grade their solutions. Instructors have also stated that this MC³ is closely related to incorrect teaching of when to use the `break` command.

Consequences This MC³ can lead to redundant complexity in the students' future code. Incomplete understanding of the `while` command can also hinder comprehension of later concepts taught in the CS1 course, such as iterating over data structures (e.g., lists and dictionaries).

Intervention Students should be instructed that the `while` condition is always verified at the end of its block. Instructors can illustrate this with simple exercises during lectures. Note that this MC³ does not apply to perpetual loops (i.e., `while True` loops) since those explicitly require conditions different than `True` to be terminated with a `break` statement.

A.1.7 C2: Redundant or unnecessary loop

Description This MC³ occurs when students declare a `while` or `for` loop that deliberately executes only once.

Example In Code A.7, the `for` declared in line 4 is executed only once.

```
1 numbers = [int(num) for num in input().split()]
2 max_num = max(numbers)
3
4 for i in range(1):
5     print(max_num)
```

Code A.7: C2: Redundant or unnecessary loop.

DIF 16

Rationale An underlying reason for this MC³ is that when students first learn about iteration commands, they believe they should use them immediately in the next assignment. This phenomenon is referred to as a *knee-jerk* reaction. Note that MC³ B6 is a special case of C2 when using `while` commands.

Consequences This MC³ can lead to redundant complexity in the students' future code. Incomplete understanding of iteration commands can also hinder comprehension of later concepts taught in the CS1 course, such as iterating over data structures (e.g., lists and dictionaries).

Intervention Students should be instructed that loops should only be used when a block of code is expected to execute more than once.

A.1.8 C4: Arbitrary number of `for` loop executions instead of `while`

Description This MC³ occurs when students declare a `for` loop to execute an excessively high number of iterations, aiming to replicate the functionality of a `while` loop.

Example In Code A.8, the `for` declared in line 3 is executed 9999 times trying to replicate a `while` loop.

```
1 numbers = []
2
3 for i in range(9999):
4     num = int(input())
5     if num == 0:
6         break
7     numbers.append(num)
```

Code A.8: C4: Arbitrary number of `for` loop execution instead of `while`.

DIF 16

Rationale Students presenting this MC³ might be deliberately preferring to use `for` loops instead of `while` loops. This preference can be related to faulty comprehensions of how `while` loops are constructed.

Consequences This MC³ leaves the code prone to failure if it is executed more times than the arbitrarily defined number of executions.

Intervention Students should be instructed about the differences between using `for` and `while` loops. Instructors can use examples to illustrate that `while` loops are generally best suited when the total number of executions is previously unknown.

A.1.9 C8: `for` loop having its iteration variable overwritten

Description This MC³ occurs when students reassign the value of the iteration variable of a `for` loop inside the loop's body.

Example In Code A.9, lines 7 and 8 are incrementing the iteration variables of the declared `for` loops from lines 4 and 1, respectively.

```
1 for i in range(1, 11):
2     print("Summation␣Table␣of", i)
3
4     for j in range(1, 11):
5         print(i, "+", j, "=", i + j)
6
7         j = j + 1
8     i = i + 1
```

Code A.9: C8: `for` loop having its iteration variable overwritten.

DIF 30

Rationale This MC³ arises because students might have difficulties understanding the concepts of the `for` command. Specifically, students are mistakenly believing they should explicitly alter the iteration variable similar to `while` loops. Another possibility is a careless approach while coding, as the students are not foreseeing the consequences of overwriting the iteration variable.

Consequences In a single `for` loop, the effect of redefining the iteration variable is negligible, as the structure simply ignores the previous redefinition. However, this MC³ can cause unexpected execution behaviors in nested loops that depend on the iteration variables from outer loops.

Intervention Students should be advised not to redefine the iteration variable of a `for` loop. Instructors can emphasize that the `for` command already increments the iteration variable at the end of a loop. Moreover, if there is a need to use the value of an iteration variable inside the loop, students should be oriented to create another variable that uses the iteration variable.

Category D: Function parameter use and scope

A.1.10 D4: Function accessing variables from outer scope

Description This MC[3] occurs when students declare functions that attempt to access variables that are not present in the function's scope.

Example In Code A.10, `res` is accessing values from variables `a` and `b` in line 2. These variables are not present in `SquaredSum()`'s scope.

```
1 def SquaredSum():
2     res = a**2 + 2*a*b + b**2
3     return res
4
5 a = int(input("Term a: "))
6 b = int(input("Term b: "))
7
8 print(SquaredSum())
```

Code A.10: D4: Function accessing variables from outer scope.

DIF 16

Rationale This MC[3] is related to a careless approach when using functions. While the general scenario may not represent a misunderstanding of function concepts *per se*, some students might be having difficulties with scope concepts.

Consequences This careless approach to coding can lead to issues regarding readability and maintainability of future code created by students.

Intervention Students should be instructed to avoid using global variables whenever possible. Instructors can provide various examples of variable scope during lectures, while also emphasizing that when a function requires outer variables, they should be passed as arguments.

Category E: Reasoning

A.1.11 E2: Redundant or unnecessary use of lists

Description This MC[3] occurs when students excessively depend on lists as a solution for various programming problems.

Example In Code A.11, `numbers` was declared as a list to read an undefined number of entries. However, since the objective was to present the sum of all entries, it could have been done while reading each value.

```
 1 numbers = []
 2 while True:
 3     num = int(input())
 4     if num == 0:
 5         break
 6     numbers.append(num)
 7
 8 sum_nums = 0
 9 for num in numbers:
10     sum_nums += num
11
12 print(sum_nums)
```

Code A.11: E2: Redundant or unnecessary use of lists.

DIF 14

Rationale This MC³ arises from students overly relying on lists as a solution for most assignments. After learning about its concepts, students tend to use this structure even when it is unnecessary for remembering or organizing a set amount of data.

Consequences Efficiency in terms of code running time is not the primary issue with this MC³. Instead, this overreliance on lists can obscure students' reasoning while solving assignments, preventing them from considering other data structures, such as dictionaries, when appropriate.

Intervention Students should be instructed to evaluate their use of lists when coding. Instructors can emphasize that these structures should be used when there is a need to remember a set of values. Moreover, instructors can also point out how exclusively using lists can lead to complex code, especially when other structures such as dictionaries could be more suitable.

Category F: Test cases

A.1.12 F2: Specific verification for instances of open test cases

Description This MC³ occurs when students create specific `if` conditions checking the input from the openly shared test cases to print the expected corresponding output.

Example Suppose that a set of input from open test cases for Code A.12 was $I = \{\{1,1,1\},\{1,3,5\},\{1,2,3,4,5\}\}$ and the expected output was $O = \{\{3\},\{9\},\{15\}\}$. The student created specific checks for each entry of I in lines 3, 5, and 7 while printing directly the expected output of O in corresponding lines 4, 6, and 8.

```
numbers = [int(num) for num in input().split()]

if numbers == [1, 1, 1]:
    print(3)
if numbers == [1, 3, 5]:
    print(9)
if numbers == [1, 2, 3, 4, 5]:
    print(15)
```

Code A.12: F2: Specific verification for instances of open test cases.

DIF 12

Rationale The main cause of this MC^3 is related to students having difficulties understanding either the assignment itself and/or how the autograder works. This can be problematic because it might appear that students are attempting to cheat the system by aiming for a minimum grade by passing only the open test cases.

Consequences Difficulties in understanding how the autograder works can hinder students' ability to reason and elaborate their solutions for future assignments. Needless to say, any attempts to cheat the system must be discouraged.

Intervention Students should be instructed that their code needs to be general in terms of the inputs, not just tailored to pass the open test cases. Instructors should provide detailed explanations of how the autograder system interacts with submitted code during lectures to clarify its functionality.

Category G: Code organization

A.1.13 G4: Functions/variables with non significant name

Description This MC^3 occurs as consequence of students using arbitrary names for variables and functions in their code.

Example In Code A.13, the naming of variables `a, b, c, d, f` and function `func()` turns the code difficult for a human to read and understand it.

```
1  a = float(input())
2  b = float(input())
3  c = float(input())
4  d = float(input())
5
6  def func(a, b, c, d):
7      f = ((c - a) ** 2  + (d - b) ** 2) ** 0.5
8      return f
9
10 print(func(a, b, c, d))
```

Code A.13: G4: Functions/variables with non significant name.

DIF 16

Rationale This MC^3 arises from a general careless approach to coding. Students often neglect to consider the significance of naming their variables, assuming their code functions adequately without meaningful identifiers. Instructors have emphasized that this programming practice may suffice for assessing simple code snippets but should never be used in the final submission for an assignment.

Consequences This MC^3 can result in the development of future code that *severely* lacks readability and maintainability.

Intervention Students should be instructed that the names of variables or functions should reflect their purpose in the code. Instructors can also provide guidance on striking a balance between commenting code and giving meaningful names to variables and functions. Examples of larger, more complex code that does not have meaningful names of variables or functions can be used to underscore the importance of readability and maintainability.

A.1.14 G5: Arbitrary organization of declarations

Description This MC^3 occurs when students structure their code arbitrarily, with functions declared interchangeably with other statements.

Example In Code A.14, the code interchanges between function definitions and variable manipulations.

```
1  def ReadData():
2      data = [int(num) for num in input().split()]
3      return data
4
5  numbers = ReadData()
6
7  def multiply(data):
8      total = 1
9      for num in data:
10         total *= num
11     return total
12
13 print(multiply(numbers))
```

Code A.14: G5: Arbitrary organization of declarations.

DIF 12

Rationale This MC³ arises from a general careless approach to coding. It often reflects students' thought processes while developing their solutions; once the code functions correctly, students may neglect to reorganize it for clarity.

Consequences This MC³ can result in the development of future code that lacks readability and maintainability.

Intervention Students should be instructed to organize all function declarations at the beginning of the code. Similarly to G4, instructors can use examples of larger, more complex code that has functions declared in arbitrary placements to emphasize the importance of readability and maintainability.

Category H: Other

A.1.15 H1: Statement with no effect

Description This MC³ occurs when students include code in their programs that has no effect on the program's execution.

Example In Code A.15, the `round()` function invoked in line 5 has no effect since its returning value was not assigned to a variable.

```
1  num1 = int(input())
2  num2 = int(input())
3
4  div_res = num1 / num2
5  round(div_res, 2)
6  print(format(div_res, ".2f"))
```

Code A.15: H1: Statement with no effect.

DIF 16

Rationale This MC³ can manifest in various forms, with the loss of a returning value not assigned to a variable as its primary cause. Sometimes, students may include lone statements such as `True` inside an unnecessary `else` clause to ensure the code functions. Generally, this MC³ indicates faulty comprehension of diverse Python constructs.

Consequences If left unaddressed, this MC³ can result in future code with increased complexity due to redundant statements, sometimes prone to errors in execution.

Intervention Students should be instructed to review their code to identify any redundant statements. Instructors can reinforce during lectures that the returning value of a function should be assigned to variables.

A.2 Other MC³

This section presents information on the other 30 MC³ that were also identified in this thesis. Since these were not further explored in research with CS1 instructors and undergraduates, information on *Rationale*, *Consequences*, and *Intervention* fields are mainly based on my conclusions after consulting CS1 instructors and on the rare occasion these MC³ showed up in the conversations with MC102 students.

Category A: Variables, identifiers, and scope

A.2.1 A1: Unused variable

Description Indicates an occurrence in which a variable was declared somewhere in the code, but it was not used later.

Example In Code A.16, `sum_vars` was declared in line 4 but was not used later.

```
1 num1 = int(input())
2 num2 = int(input())
3
4 sum_vars = num1 + num2
5
6 print(num1 + num2)
```

Code A.16: A1: Unused variable.

DIF -6

Rationale This MC³ arises from students' lack of attention while coding. A possible scenario is when students change their strategy to solve an assignment but do not fully erase their previous code.

Consequences This type of inattention can lead to neglecting important coding aspects such as readability and maintainability.

Intervention Students should be instructed to revise their code before making their final submission. Instructors and teaching assistants can also point out how different programming environments highlight unused code, for example, by changing the color of the font.

A.2.2 A2: Variable assigned to itself

Description Indicates an occurrence in which a variable receives itself as a result of an assignment.

Example In Code A.17, `odd` is assigned to itself in line 7.

```
1 num1 = int(input())
2
3 odd = False
4 if num1 % 2 == 1:
5     odd = True
6 else:
7     odd = odd
8
9 print("odd?:", odd)
```

Code A.17: A2: Variable assigned to itself.

DIF 8

Rationale This MC[3] arises possibly due to confusion about how variable assignments work. In Code A.17, students might think that the variable loses its value even when the condition for it to be changed is not met. Other scenarios include redundant code explicitly declared in an unnecessary `else` clause. In this case, students may mistakenly believe `else` is mandatory and therefore create a redundant statement to avoid altering the code's output.

Consequences This misconception about assignments can lead to the development of more redundant code in the future, resulting in poor readability and maintainability.

Intervention Students should be instructed on how variables can have their assigned values changed when using different control structures in the code. Simple exercises can be presented in lectures to demonstrate a variable's output through different code executions.

A.2.3 A3: Variable unnecessarily initialized

Description Indicates an occurrence in which a variable is unnecessarily initialized before being used in the code.

Example In Code A.18, `var1` and `var2` are assigned arbitrary values before receiving the intended results from `input()` in lines 3 and 4, respectively.

```
1 var1 = 0
2 var2 = ''
3 var1 = int(input())
4 var2 = input()
```

Code A.18: A3: Variable unnecessarily initialized.

DIF -8

Rationale This MC[3] arises possibly due to confusion about how the assignment of variables works. A possible scenario is that students may mistakenly believe every variable must be assigned a dummy value before receiving an external input. This scenario could also be related to prior programming education with strongly typed languages, such as C.

Consequences This misconception about assignments can lead to the development of more redundant code in the future, resulting in poor readability and maintainability.

Intervention Instructors can reinforce that `input()` overwrites any preexisting value of a variable, emphasizing that the assignment of dummy values is redundant.

A.2.4 A5: Unused import

Description Indicates an occurrence in which an import from a library is declared in the code, but none of its functionalities are used.

Example In Code A.19, the `math` module was imported but was not used.

```
1 import math
2 base = 3
3 exp = 4
4
5 print(base**exp)
```

Code A.19: A5: Unused import.

DIF -22

Rationale This MC³ arises from students' lack of attention while coding. It mainly stems from reusing code snippets that instructors might have used during lectures or a piece of code that was previously removed.

Consequences This MC³ can easily be removed from the code. However, if not removed, it can limit the portability of the code since other environments might not have the required libraries.

Intervention Instructors should explain how and why imports are needed in the code. Instructors and teaching assistants can also point out that some programming environments detect unused imports, for example, by underlining them with a different color.

A.2.5 A6: Variables with arbitrary values (Magic Numbers) used in operations

Description Indicates an occurrence in which arbitrary values are used in the code. These values, also known as "magic numbers", are reliant on the problem instance and can manifest either in the form of a lone variable or hard coded into another variable.

Example In Code A.20, the value of `total` is being hard coded with a magic number in line 2.

```
1 price = float(input())
2 total = price * 0.15
3
4 print(total)
```

Code A.20: A6: Variables with arbitrary values (Magic Numbers) used in operations.

DIF 8

Rationale One possible underlying reason for this MC³ is that students might lack confidence in declaring and manipulating different variables.

Consequences This MC³ can lead to issues regarding maintainability in future code created by the students.

Intervention Students should be instructed that magic numbers can be avoided by assigning these numbers to constants at the beginning of the code.

A.2.6 A7: Arbitrary manipulations to modify declared variables

Description Indicates an occurrence in which variables have their values altered by arbitrary means, such as erasing its value by assigning an empty string to it.

Example In Code A.21, `var` is being assigned an empty string in line 4 to erase its previous value before being assigned a new one in line 5.

```
1 var = int(input())
2 while var != -1:
3     (...)  #some operation with current var value
4     var = ''
5     var = int(input())
```

Code A.21: A7: Arbitrary manipulations to modify declared variables.

DIF 8

Rationale This MC³ arises possibly due to students having difficulties with the manipulation of variables.

Consequences The arbitrary manipulation of variables in this MC³ can lead to issues regarding readability and maintainability in future code created by the students.

Intervention Students should be instructed that every direct assignment overwrites the previous value of a variable.

A.2.7 A8: Arbitrary treatment of the stopping point of reading values

Description Indicates an occurrence in which the stopping condition for reading data is not correctly implemented. In this scenario, the guard condition is included in the set of valid data.

Example In Code A.22, the guard condition `-1` is also appended to `numbers` before the loop execution is terminated.

```
1  numbers = []
2  while True:
3      var = int(input())
4      numbers.append(var)
5      if var == -1:
6          break
7
8  for num in numbers:
9      if num == -1:
10         numbers.remove(num)
```

Code A.22: A8: Arbitrary treatment of the stopping point of reading values.

DIF 4

Rationale The underlying reason for this MC³ is a careless approach students have while coding.

Consequences Leaving invalid entries in the stored data can lead to unexpected code executions if students forget to remove these invalid entries.

Intervention Students should be instructed not to include invalid entries in an expected valid data set.

Category B: Boolean expressions

A.2.8 B1: Redundant or simplifiable Boolean comparison

Description Indicates an occurrence in which a Boolean comparison can be expressed in a simpler way.

Example In Code A.23, the left hand side of the expression in line 4 is redundantly being compared to `True`.

```
1 num1 = int(input())
2 num2 = int(input())
3
4 if (num1 >= num2) == True:
5     print(num1, "greater than or equal to", num2)
6 else:
7     print(num1, "lesser than", num2)
```

Code A.23: B1: Redundant or simplifiable Boolean comparison.

DIF -8

Rationale This MC³ arises from students who are still grasping the fundamental concepts of decision statements.

Consequences This MC³ indicates a potential misunderstanding of conditional commands if students are still routinely making this mistake by the end of the CS1 course.

Intervention Instructors can reinforce the concepts of redundant Boolean comparisons in lectures using simple examples.

A.2.9 B2: Boolean comparison separated in intermediary variables

Description Indicates an occurrence in which a boolean comparison, with more than two factors, is coded by assigning the preliminary results in auxiliary variables, instead of calculating the whole expression in the same operation.

Example In Code A.24, `cond1` and `cond2` were declared as intermediary values for the test conducted in line 8.

```
1 age = int(input())
2 c_time = int(input())
3 gender = input()
4
5 cond1 = age > 65
6 cond2 = c_time > 10
7
8 if cond1 and cond2 and gender == 'M':
9     print("Can retire.")
```

Code A.24: B2: Boolean comparison separated in intermediary variables.

DIF -18

Rationale This MC[3] is only a real issue if students mistakenly believe that conditional statements can only be constructed using variables. Aside from that, separating code in auxiliary variables can keep the conditional statements short and more readable.

Consequences This MC[3] indicates a potential misunderstanding of conditional commands if students mistakenly believe that conditional statements can only be constructed using variables.

Intervention Students should be instructed that breaking long expressions into variables is a good idea, but only when those variables are significantly named.

A.2.10 B3: Arithmetic expression instead of Boolean

Description Indicates an occurrence in which a Boolean expression is instead defined as an arithmetic expression. In this case, students assign numbers to certain simpler conditions and then perform an operation with these conditions, such as the sum of them.

Example In Code A.25, the decision statement declared in line 7 uses the sum of two Boolean variables as its condition.

```
1 num1 = int(input())
2 num2 = int(input())
3
4 cond1 = num1 % 2 == 0
5 cond2 = num2 % 2 == 0
6
7 if cond1 + cond2 == 2:
8     print("odd numbers")
```

Code A.25: B3: Arithmetic expression instead of Boolean.

DIF 6

Rationale This MC³ arises from a possible misunderstanding of Boolean variables, leading to discomfort in using them with Boolean operators. This misconception might occur when students learn that, in Python, `True` can be interpreted as `1` and `False` can be interpreted as `0`.

Consequences This MC³ adds unnecessary complexity to the code's logic, leading to issues regarding readability and maintainability of future code.

Intervention Students should be instructed to avoid treating Boolean variables as integers whenever possible.

A.2.11 B4: Repeated commands inside `if-elif-else` blocks

Description Indicates an occurrence in which the same commands appear in different blocks of `if-elif-else`.

Example In Code A.26, lines 5 to 7 and 10 to 12 are equal and could be factorized out of the `if-else` statement.

```
1  order = input()
2
3  if order == "online":
4      print("Processing␣online␣order")
5      print("Calculating␣shipping␣costs...")
6      print("Applying␣discount...")
7      print("Finalizing␣order...")
8  else:
9      print("Processing␣in-store␣order")
10     print("Calculating␣shipping␣costs...")
11     print("Applying␣discount...")
12     print("Finalizing␣order...")
```

Code A.26: B4: Repeated commands inside `if-elif-else` blocks.

DIF 6

Rationale Although this MC³ can be traced to a lack of attention while coding, these repeating commands are often the result of copying and pasting code.

Consequences This MC³ leaves the code prone to errors due to the act of copying and pasting. Additionally, it can lead to future issues with code maintenance, as students may forget to alter all repeating code sequences when necessary.

Intervention Instructors can explain in lectures that repeating code in these situations can be refactored by moving them outside of `if-elif-else` blocks.

A.2.12 B5: Nested `if` statements instead of Boolean comparison

Description Indicates an occurrence in which boolean comparisons are coded by nesting `if` statements instead of using logic operators.

Example In Code A.27, lines 4 and 5 could have been merged in a single `if` by using Boolean operators.

```
1 num1 = int(input())
2 num2 = int(input())
3
4 if num1 > 0:
5     if num2 > 0:
6         print(num1, "positive", num2, "positive")
```

Code A.27: B5: Nested `if` statements instead of Boolean comparison.

DIF -10

Rationale If there are only a few variables, this behavior might not be an issue. However, this MC^3 arises if there are more variables handled in this manner, as this potentially indicates students are avoiding using Boolean operators, possibly due to misunderstandings.

Consequences If multiple variables are involved, this MC^3 can lead to future complex code created by the students. This can negatively influence the code's readability and maintainability.

Intervention If instructors or teaching assistants detect students routinely nesting multiple `if` statements, they can address merging using Boolean expressions with simple examples during lectures.

A.2.13 B7: Boolean validation variable instead of `elif/else`

Description Indicates an occurrence in which a Boolean variable (*validation flag*) is declared and assigned a condition that is tested further with consecutive `if` statements when an `elif/else` could be implemented without the need of this flag.

Example In Code A.28, the `flag` variable declared in line 1 is used in sequential `if` statements in lines 7, 9, and 11. The statements in lines 9 and 11 could have been declared as `elif-else` blocks.

```
1  flag = False
2  var = input()
3
4  if len(var) == 0:
5      flag = True
6
7  if flag:
8      print("empty␣message")
9  if not flag and len(var) > 1:
10     print("words:", var.split())
11 if not flag and len(var) == 1:
12     print("word:", var)
```

Code A.28: B7: Boolean validation variable instead of `elif/else`.

DIF 4

Rationale Although flag variables are commonly taught in CS1 classes, students might mistakenly use them to structure their code only with `if` statements. This preference over `elif-else` statements can stem from misunderstandings about these constructs.

Consequences This MC[3] can lead to future complexity in code, resulting in maintenance issues due to the use of flag variables.

Intervention Students should be instructed on the proper usage of flag variables. Instructors can present clearly constructed examples during lectures to demonstrate when these variables are most appropriate.

A.2.14 B10: Unnecessary `elif/else`

Description Indicates an occurrence in which `elif/else` statements are declared but their blocks contain code that has no effect.

Example In Code A.29, the student checked all possible cases for `num` in lines 2 and 4, while also including an unnecessary `else` statement in line 6. In this case, the `else` could have been properly used in line 4.

```
1  num = int(input())
2  if num >= 0:
3      print(num, "is␣positive␣or␣0")
4  elif num < 0:
5      print(num, "is␣negative")
6  else:
7      (...) #some code that does not alter the output
```

Code A.29: B10: Unnecessary `elif/else`.

DIF 6

Rationale This MC³ arises because students might be having difficulties with conditional statements, mistakenly believing that `elif/else` statements are mandatory.

Consequences This MC³ can lead to future redundant code, causing issues with readability and maintainability due to misunderstandings of conditional statements.

Intervention Students should be oriented that `elif/else` statements are not mandatory and should only be used when appropriate.

A.2.15 B11: Consecutive distinct `if` statements with the same operations in their blocks

Description Indicates an occurrence in which consecutive `if` statements that check distinct conditions are declared, but there are repeated commands in their blocks.

Example In Code A.30, lines 5, 7, and 9 could have been merged into a single `if` since their blocks performs the same commands.

```
1  e1 = float(input("Edge␣1:␣"))
2  e2 = float(input("Edge␣2:␣"))
3  e3 = float(input("Edge␣3:␣"))
4
5  if e1 <= 0:
6      print("invalid␣input")
7  if e2 <= 0:
8      print("invalid␣input")
9  if e3 <= 0:
10     print("invalid␣input")
```

Code A.30: B11: Consecutive distinct `if` statements with the same operations in their blocks.

DIF 8

Rationale This MC³ arises because students might be having difficulties with conditional statements, as the commands in the distinct `if` statements could be grouped into a single statement.

Consequences While there are situations where this coding pattern is necessary, generally, this MC³ stems from a lack of clarity and may result in future code quality issues, such as readability and maintainability.

Intervention Students should be encouraged to assess their code to identify possible refactoring opportunities that will enhance readability and maintainability.

Category C: Iteration

A.2.16 C3: Redundant operations inside loop

Description Indicates an occurrence in which operations are being unnecessarily calculated inside a loop.

Example In Code A.31, `average` is being calculated repeatedly and unnecessarily in line 7, since this operation could have been performed once after the loop.

```
1 numbers = [int(i) for i in input().split()]
2 sum_num = total_num = 0
3
4 for num in numbers:
5     sum_num += num
6     total_num += 1
7     average = sum_num / total_num
8
9 print(average)
```

Code A.31: C3: Redundant operations inside loop.

DIF 10

Rationale This MC³ arises because students might be having difficulties identifying refactoring possibilities within their code.

Consequences This MC³ can lead to increased code complexity and inefficiency. However, teaching about efficiency attributes such as code running time and resource consumption is debatable among CS1 instructors.

Intervention If efficiency in resource consumption is within the learning objectives of the CS1 course, students should be encouraged to refactor their code to identify and remove redundant operations.

A.2.17 C5: Use of intermediary variables to loop control

Description Indicates an occurrence in which the variable that controls the loop is updated via other intermediary variables which are fuzzy or obfuscated.

Example In Code A.32, `var1` is updated using intermediary variables `aux` and `var2` to determine the execution of the `while` loop.

```
1 var1 = int(input())
2 var2 = 1
3 while var1 != 0:
4     aux = int(input())
5     var1 = aux + var2
6     (...)
```

Code A.32: C5: Use of intermediary variables to loop control.

DIF -12

Rationale This MC³ arises due to a lack of clarity in students' thought processes, reflecting in a lack of attention while coding.

Consequences Obfuscated logic in manipulating a loop's conditional variable can lead to issues in readability and maintainability of future code.

Intervention Similar to MC³ B6, students should be oriented to divide Boolean expressions into Boolean variables only if these variables have significant names.

A.2.18 C6: Multiple distinct loops that operates over the same iterable

Description Indicates an occurrence in which multiple consecutive loops that operate over the same iterable are declared with different operations being performed inside their respective blocks.

Example In Code A.33, three `for` loops that iterates over `numbers` were declared in lines 4, 8, and 12. All loops could have been merged into a single one.

```
1  numbers = [int(i) for i in input().split()]
2  sum_odd = sum_even = sum_num = 0
3
4  for num in numbers:
5      if num % 2 == 1:
6          sum_odd += num
7
8  for num in numbers:
9      if num % 2 == 0:
10         sum_even += num
11
12 for num in numbers:
13     sum_num += num
```

Code A.33: C6: Multiple distinct loops that operates over the same iterable.

DIF 0

Rationale This MC³ arises because students might have difficulties identifying refactoring possibilities within their code. In this case, students are possibly dividing their logic into multiple steps and overlook refactoring once the code produces the expected outcome.

Consequences Similar to MC³ C3, this MC³ can lead to increased code complexity in terms of efficiency. However, teaching about efficiency attributes such as code running time and resource consumption is debatable among CS1 instructors.

Intervention As with MC³ C3, if efficiency in resource consumption is within the learning objectives of the CS1 course, students should be encouraged to assess their code to identify potential sections that can increase performance.

A.2.19 C7: Arbitrary internal treatment of loop boundaries

Description Indicates an occurrence in which the body of a loop contains specific conditionals treating its boundary values.

Example In Code A.34, two `if` statements (lines 4 and 7) were declared to perform operations inside the `for` loop declared in line 3. However, since these `if` statements treats the boundary values, they could have been factored out of the loop.

```
1  numbers = [int(i) for i in input().split()]
2
3  for iter in range(len(numbers)):
4    if iter == 0:
5      (...)
6
7    if iter == len(numbers) - 1:
8      (...)
9
10   (...)
```

Code A.34: C7: Arbitrary internal treatment of loop boundaries.

DIF 2

Rationale This MC³ arises because students might have difficulties identifying refactoring possibilities within their code. In this case, students are overlooking the fact that the boundary checks could have been performed outside the loop.

Consequences Similar to previous behaviors, this MC³ can lead to increased code complexity in terms of efficiency. However, teaching about efficiency attributes such as code running time and resource consumption is debatable among CS1 instructors.

Intervention If efficiency in resource consumption is within the learning objectives of the CS1 course, students should be encouraged to assess their code to identify potential sections that can increase performance.

Category D: Function parameter use and scope

A.2.20 D1: Inconsistent `return` declaration

Description Indicates an occurrence in which only some of the `return` declarations inside a function return an expression.

Example In Code A.35, there should have been another `return` considering the case the `if` statement in line 2 is evaluated as `False`.

```
1 def IsEven(num):
2     if num % 2 == 0:
3         return True
```

Code A.35: D1: Inconsistent `return` declaration.

DIF 6

Rationale This MC^3 arises due to students' lack of attention while coding. Specifically, students are oblivious to code conventions regarding functions.

Consequences This MC^3 can lead to the creation of code that is not formatted according to PEP8[1]: if any `return` statement returns an expression, any `return` statements where no value is returned should explicitly state this as `return None`, and an explicit `return` statement should be present at the end of the function (if reachable).

Intervention If code convention is within the learning objectives of the CS1 course, instructors should orient about the importance of following these conventions.

A.2.21 D2: Too many `return` declarations inside a function

Description Indicates an occurrence in which a function has a high number of `return` declarations inside its scope.

Example In Code A.36, function `checker()` was created with five `return` statements. This could potentially indicate that said function is large and could benefit from being split in smaller functions.

[1] `https://www.python.org/dev/peps/pep-0008/`

```
1  def process_number(n):
2      if n < 0:
3          return "Negative number"
4      if n == 0:
5          return "Zero"
6      if n > 0 and n <= 10:
7          return "Positive number less than or equal to 10"
8      if n > 10 and n <= 100:
9          return "Positive number between 11 and 100"
10     if n > 100:
11         return "Positive number greater than 100"
12
13     return "Not a number"
```

Code A.36: D2: Too many `return` declarations inside a function.

DIF -8

Rationale An underlying reason for this MC³ is a misunderstanding of modularization concepts. In this case, students are overlooking the fact that creating one large function does not assist in code modularization.

Consequences This MC³ can lead to the creation of future code that has issues in readability and maintainability.

Intervention While it may not always be the case, students should be oriented that too many `return` declarations can indicate that a function may benefit from being split into smaller functions.

A.2.22 D3: Redundant or unnecessary `return` declaration

Description Indicates an occurrence in which a `return` declaration that has no impact over the execution of the program is present inside a function.

Example In Code A.37, an unnecessary `return` was declared in line 4.

```
1  def PrintBySide(elements)
2      for item in elements:
3          print(item, end=" ")
4      return
```

Code A.37: D3: Redundant or unnecessary `return` declaration.

DIF -12

Rationale This MC³ arises due to misunderstandings of function concepts. In this case, students might mistakenly believe that the `return` statement is mandatory.

Consequences The redundancy resulting from this MC³ can lead to the creation of future code that has issues in readability and maintainability.

Intervention Students should be oriented that the `return` statement is not mandatory and should be used only when necessary. Instructors can explain this in parallel with MC³ D1.

Category E: Reasoning

A.2.23 E1: Checking all possible combinations unnecessarily

Description Indicates an occurrence in which all possible combinations of Boolean variables and/or expressions are unnecessarily declared in order to check all of their possible combinations.

Example In Code A.38, the whole truth table was unnecessarily declared to assess `cond1` and `cond2`.

```
1  cond1 = (...) #a boolean expression
2  cond2 = (...) #another boolean expression
3
4  if not cond1 and not cond2:
5      print(False)
6  if not cond1 and cond2:
7      print(False)
8  if cond1 and not cond2:
9      print(False)
10 if cond1 and cond2:
11     print(True)
```

Code A.38: E1: Checking all possible combinations unnecessarily.

DIF 10

Rationale This MC³ arises due to students having difficulties with conditional statements. In this case, students might be struggling to understand how Boolean operators work.

Consequences This MC³ can lead to the creation of redundant code, promoting issues with readability and maintainability. Moreover, as the number of variables increases, the code becomes more susceptible to errors due to the necessity of more conditions to be checked.

Intervention Students should be oriented about Boolean operators in Python. Examples can be used in lectures to address and exemplify why there is no need to assess all possible combinations of Boolean variables.

Category F: Test cases

A.2.24 F1: Verification for non explicit conditions

Description Indicates an occurrence in which the student over verifies conditions for the code, which were guaranteed by the assignment description they will not be present in the test cases.

Example Consider that the assignment description for Code A.39 explicitly stated that all test cases would consider the ages as greater than 0. Even so, the student assessed this given information in line 4.

```
1 majorityAge = int(input())
2 age = int(input())
3
4 if age >= majorityAge and age > 0 and majorityAge > 0:
5     print("Adult")
```

Code A.39: F1: Verification for non explicit conditions.

DIF 0

Rationale A sense of safe programming is not an issue *per se*. This MC^3 can only be a problem if the student decides to unnecessarily check superfluous conditions that leaves the code prone to errors.

Consequences The redundancy resulting from this MC^3 can lead to the creation of future code that has issues in readability and maintainability. A lack of attention while creating unnecessary checks can leave the code prone to errors.

Intervention Although preventing unexpected input is a good programming practice, students should also be oriented to understand the assignment's description regarding the expected functionality of the code.

Category G: Code organization

A.2.25 G1: Long line commentary

Description Indicates an occurrence in which long commentaries are written in the line format in Python, extending the file horizontally.

Example In Code A.40, a long commentary was inserted in the line format (line 1).

```
1 (...) #a long line commentary that thoroughly explains this line of
      code but the block format would have been more appropriate
      instead...
```

Code A.40: G1: Long line commentary.

DIF -18

Rationale This MC^3 arises due to an incomplete understanding of Python properties. In this case, students are unfamiliar with block comment formats.

Consequences As this coding behavior mostly adheres to a programming style, this MC^3 would only be an issue if the student is not following a predetermined style within the CS1 course. However, a possibly related problem would reside in students' reasoning, as they might create long explanations for code instead of summarizing them in shorter comments.

Intervention Students should be oriented about the proper usage of line and block comments in Python. Additionally, instructors can reinforce that significant variable and function names can reduce the amount of comments needed in the code.

A.2.26 G2: Exaggerated use of variables to assign expressions

Description Indicates an occurrence in which a high number of variables are declared as a result of simpler expressions. These variables are later used in the code.

Example In Code A.41, a high number of variables were assigned simple expressions.

```
1  a, b, c = float(input()), float(input()), float(input())
2  d = b ** 2
3  e = c ** 2
4  f = a ** 2
5  g = f - (e + d)
6  q = e - (f + d)
7  h = a + b + c
8  i = (c * b) - (a * c)
9  o = (a * b) - (a * c)
10
11 if h == 3 * a :
12     print("Equilateral␣triangle")
13 elif i == 0 :
14     print("Isosceles␣triangle")
15 elif o == 0:
16     print("Isosceles␣triangle")
17 else:
18     print("Scalene␣triangle")
```

Code A.41: G2: Exaggerated use of variables to assign expressions.

DIF -10

Rationale This MC³ arises due to students' thought processes while solving an assignment.

Consequences As previously discussed, dividing longer expressions into auxiliary variables can enhance code readability if the variables have significant names. However, if the names are not meaningful, the opposite effect can occur, negatively impacting readability and maintainability.

Intervention Students should be oriented about proper assignment when shortening expressions into auxiliary variables. Instructors should emphasize the importance of using descriptive names for auxiliary variables to maintain code clarity.

A.2.27 G3: Too many declarations in a single line of code

Description Indicates an occurrence in which many operations, distinct or not, are declared in the same line of code.

Example In Code A.42, a `for` loop and a ternary `if-else` statement were declared together in line 3.

```
1 numbers = [int(i) for i in input().split()]
2
3 for num in numbers: print(num*2) if num % 2 == 0 else print(num*3)
```

Code A.42: G3: Too many declarations in a single line of code.

DIF 10

Rationale This MC³ arises due to a preference in programming style. Students might choose to write their code this way as they might be thinking that this structure achieves efficiency in resource consumption by concatenating commands in a single line of code.

Consequences This MC³ can make the code prone to bugs and negatively impact readability and maintainability.

Intervention Students should be oriented to avoid writing code in this manner. Instructors can present examples during lectures that illustrate the difficulties in reading and maintaining code written with this MC³.

A.2.28 G6: Functions not documented in the Docstring format

Description Indicates an occurrence in which the declared functions are either not properly documented, or documented at all using Python Docstring format.

Example In Code A.43, the function `distance()` was briefly documented in line 2, but this comment was not formatted in Python Docstring.

```
1 def distance(x1, x2, y1, y2):
2     #calculates the Euclidean distance
3     dist = ((y1 - x1) ** 2  + (y2 - x2) ** 2) ** 0.5
4     return dist
```

Code A.43: G6: Functions not documented in the Docstring format.

DIF -4

Rationale This MC³ arises due to an incomplete understanding of Python properties. Similar to MC³ G1, students are unfamiliar with the block format commentaries used in Python Docstring.

Consequences As with MC³ G1, this coding behavior adheres to a programming style. Therefore, this MC³ would only be an issue if the student is not following a predetermined style within the CS1 course. Undocumented functions can lead to issues regarding the readability and maintainability of future code.

Intervention Students should be oriented about the existence and usage of Python Docstring. Instructors can encourage Docstring usage by presenting their advantages through examples during lectures.

Category H: Other

A.2.29 H2: Redundant typecast

Description Indicates an occurrence in which a typecast is applied to the result of a function or expression that already results in the same type.

Example In Code A.44, the typcasts performed in lines 1 and 2 were redundant as the operations already return values as `str` and `int`, respectively.

```
1 var1 = str(input())
2 var2 = int(6 + 4)
```

Code A.44: H2: Redundant typecast.

DIF 0

Rationale This MC³ arises from misunderstandings about basic Python functionalities. Specifically, students are oblivious to the resulting type of basic operations or built-ins.

Consequences The redundancy resulting from this MC³ can lead to the creation of future code that has minor issues in readability and maintainability.

Intervention Instructors can reinforce the resulting type of operations or built-ins during lectures. Moreover, students can be oriented that programming environments offer ways to determine the resulting type of an operation, such as by hovering the mouse cursor over it.

A.2.30 H3: Unnecessary or redundant semicolon

Description Indicates an occurrence in which a semicolon is used unnecessarily, typically at the end of a statement or declaration in a line of code.

Example In Code A.45, redundant semicolons were placed in lines 1 and 2.

```
1 var1 = input();
2 var2 = int(input());
3
4 print(var1, ":␣", var2, sep="")
```

Code A.45: H3: Unnecessary or redundant semicolon.

DIF -16

Rationale This MC³ arises from minor misunderstandings of Python syntax, likely because students have previous programming knowledge of other languages.

Consequences The redundancy resulting from this MC³ can lead to the creation of future code that has minor issues in readability and maintainability.

Intervention This MC³ should disappear as students progress in the CS1 course. However, instructors or teaching assistants can address this redundancy if they notice it in students' code.

Appendix B

Developed Educational Materials

This appendix presents information on the developed educational materials to address MC³. In total, four different interventions were created: automated MC³ detection tool[1], short videos (60 seconds), digital flashcards, and lecture slides[2].

B.1 Flashcards

All flashcards were developed in Brazilian Portuguese and are listed below.

B.1.1 A4: Redefinition of built-in

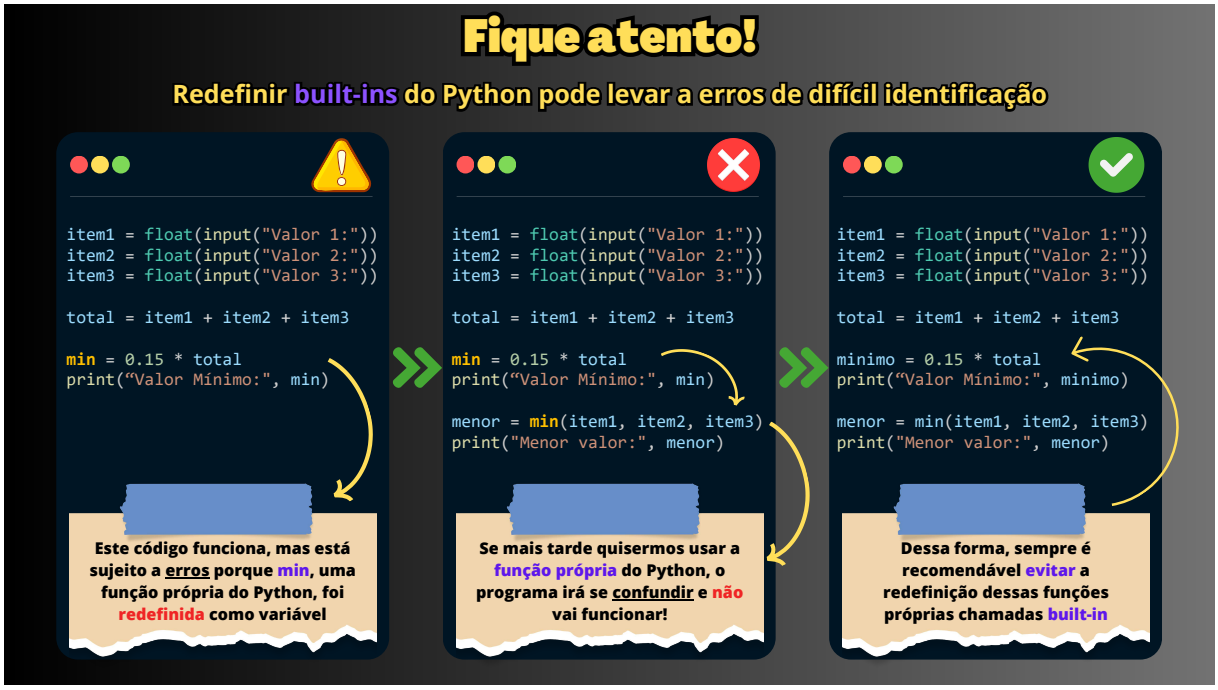

Figure B.1: A4: Redefinition of built-in.

[1] https://github.com/eryckpedro/mc4

[2] https://drive.google.com/drive/folders/1fA-TkhGJ3Ff4KiH_K3bsywRyZPGfUJ7L?usp=drive_link

B.1.2 B9: `elif/else` retesting already checked conditions

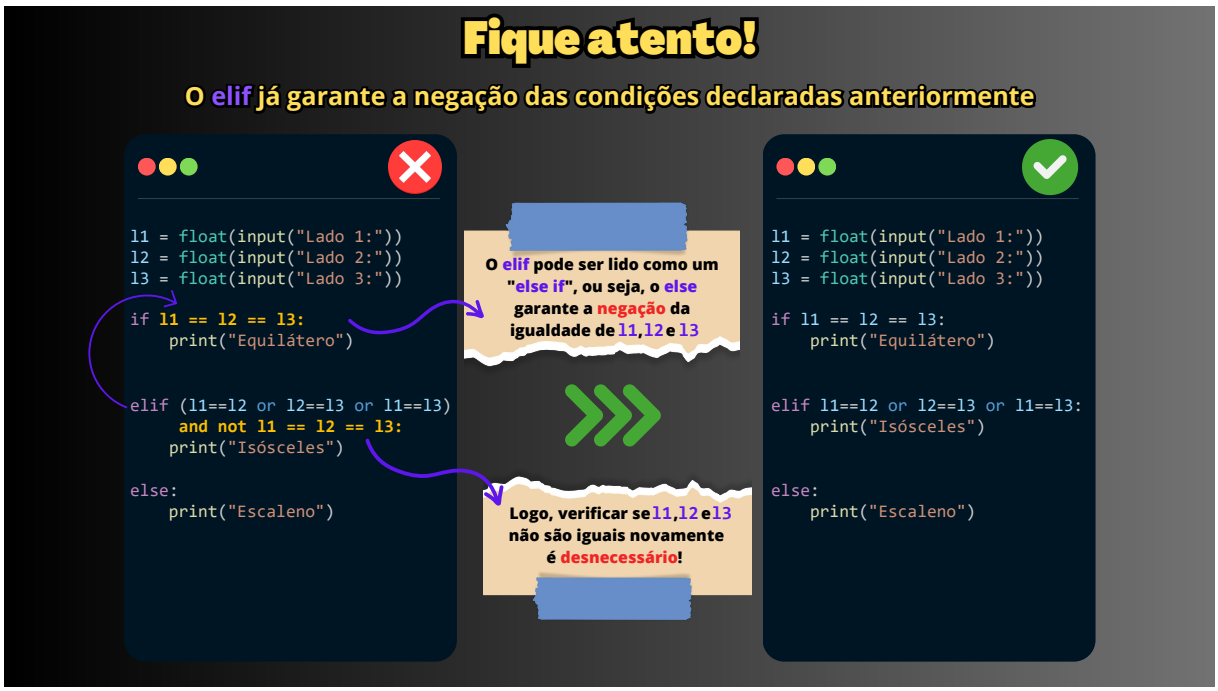

Figure B.2: B9: `elif/else` retesting already checked conditions.

B.1.3 C1: `while` condition tested again inside its block

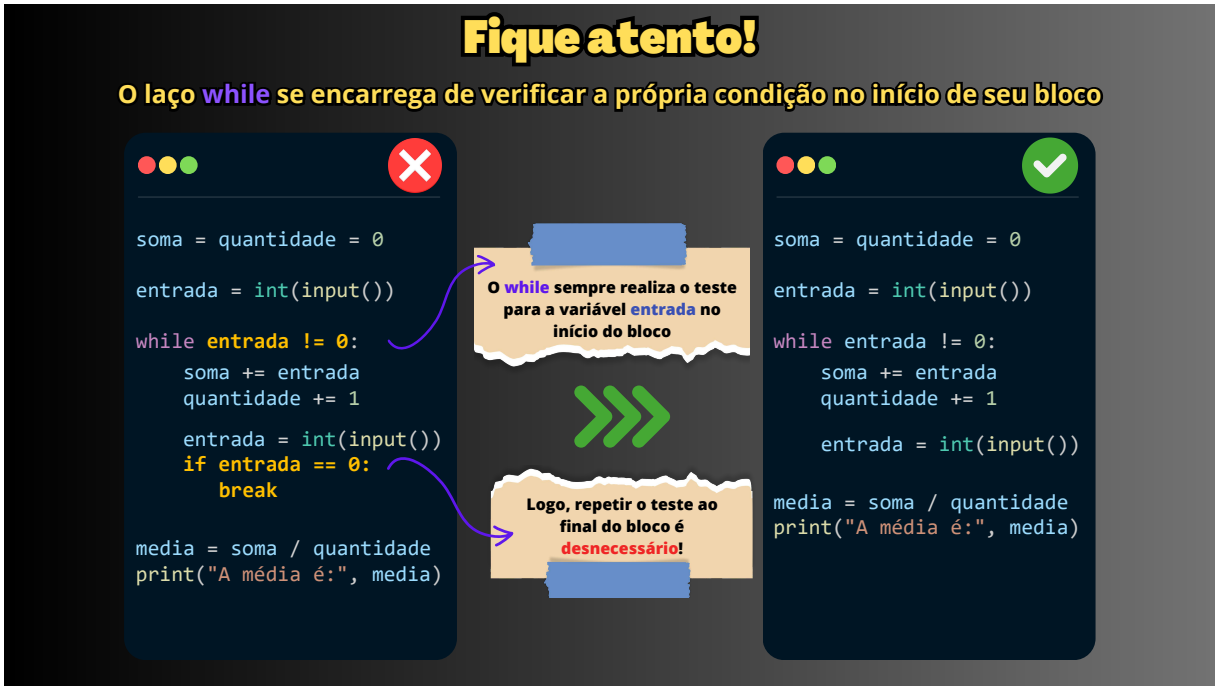

Figure B.3: C1: `while` condition tested again inside its block.

B.1.4 C8: `for` loop having its iteration variable overwritten

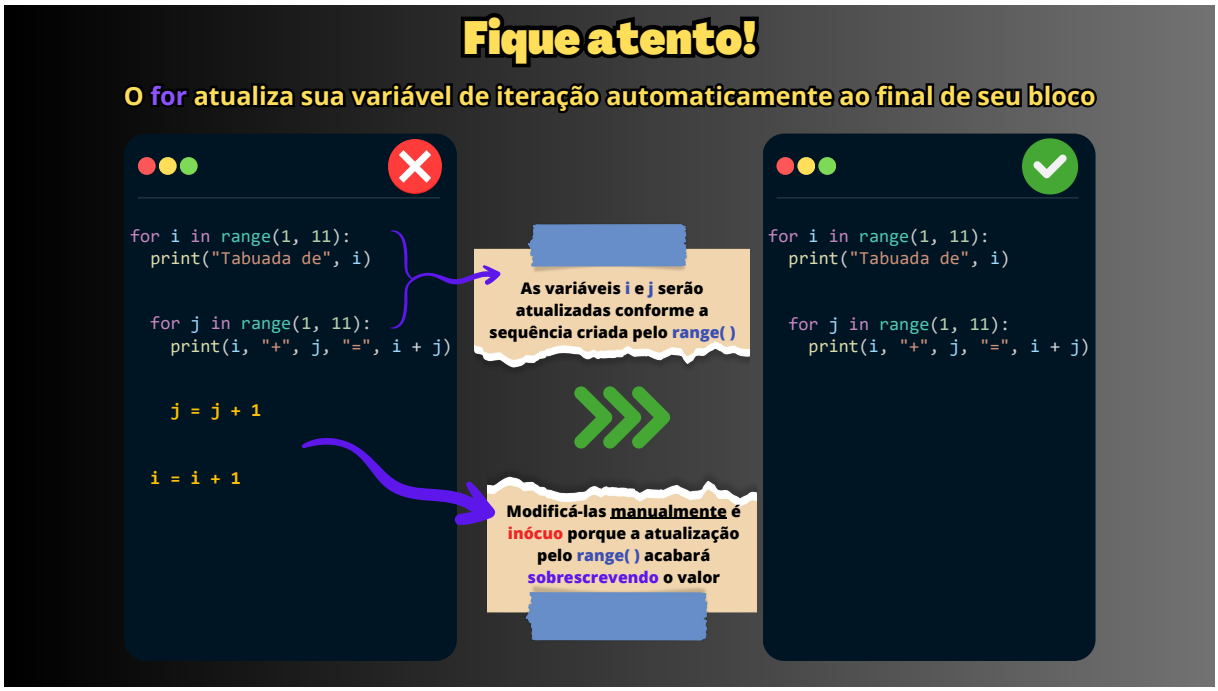

Figure B.4: C8: `for` loop having its iteration variable overwritten.

B.1.5 D4: Function accessing variables from outer scope

Figure B.5: D4: Function accessing variables from outer scope.

B.1.6 G5: Arbitrary organization of declarations

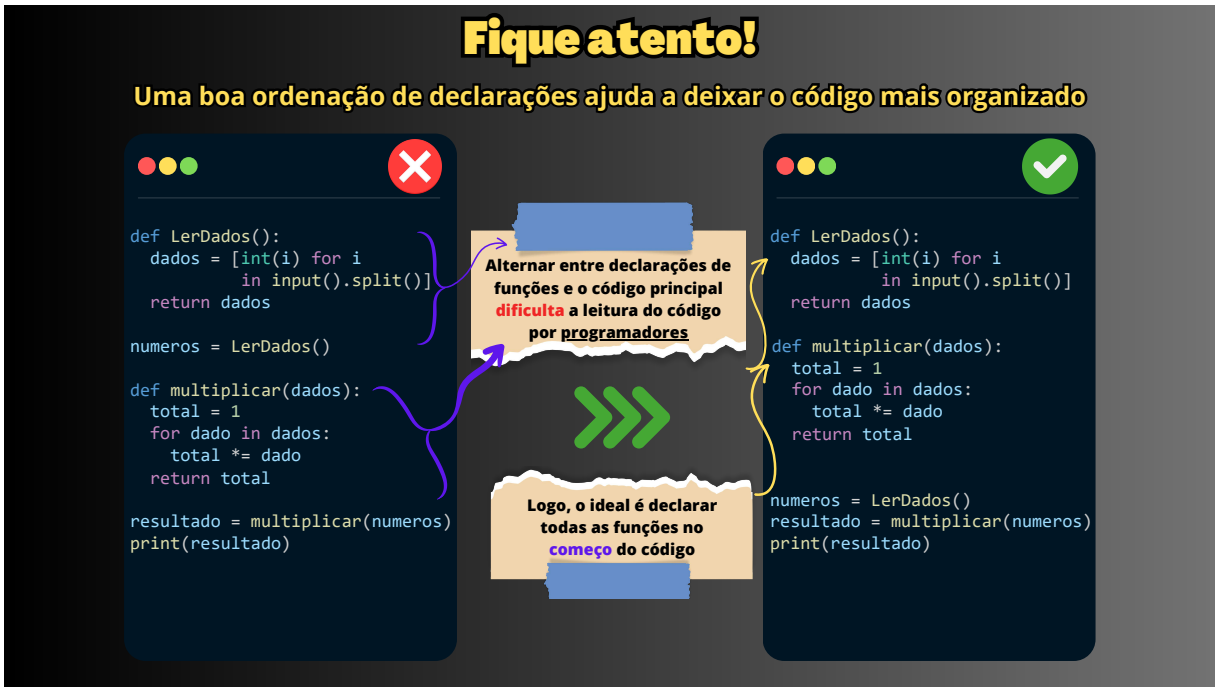

Figure B.6: G5: Arbitrary organization of declarations.

B.2 Short Videos

The short videos were also uploaded in YouTube via the Shorts format. The following list presents each MC³ with its corresponding video (in Brazilian Portuguese).

- A4: Redefinition of built-in:
  `https://youtube.com/shorts/GDV8-HA9PZg`
- B9: `elif/else` retesting already checked conditions:
  `https://youtube.com/shorts/Y7sugUt8OdQ`
- C1: `while` condition tested again inside its block:
  `https://youtube.com/shorts/wG5GJjmvvTs`
- C8: `for` loop having its iteration variable overwritten:
  `https://youtube.com/shorts/3Zq8i3U1DE4`
- D4: Function accessing variables from outer scope:
  `https://youtube.com/shorts/1O_sRXEQ6PA`
- G5: Arbitrary organization of declarations:
  `https://youtube.com/shorts/gONVV17oqtc`

Appendix C

Publisher's Authorization

As required by Universidade Estadual de Campinas for thesis written in the article collection format, this appendix presents the publisher's authorization for reuse of the papers described in Chapters 3 and 4.

Brazilian Journal of Computers in Education (RBIE)

- Source: `https://sol.sbc.org.br/journals/index.php/rbie/about/submissions`
- Accessed on May 14th, 2024.

editing

187

CRediT authorship statement example:

RDA: Conceptualization, Methodology, Writing – original draft; **AAFB:** Conceptualization, Investigation, Writing – original draft, Supervision.

Read more information about taxonomy here.

Copyright Notice

Authors who publish in this journal agree to the following conditions:

a. Authors keep their copyrights and grant the journal the publication rights, with the work simultaneously licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND). Then, it is allowed to share the work identifying its authors and referencing the publication in this journal.
b. Authors have the authorization to take on additional contracts separately, for non-exclusive distribution of the work published in this journal (ex.: publish in an institutional repository, as a book chapter, scientific magazines, etc), referencing the authors and the publication in this journal.
c. Authors have permission and are encouraged to publish and distribute their works online (in institutional repositories or in their own web page) in any moment after the editorial process, since it may generate productive altering, as well as increasing the impact and quotation of the published work (See The Free Access Effect). However, the information about the publication (journal title, volume/issue, pages) must be available.

Privacy Statement

The names and emails informed in this journal will be used exclusively for the services provided by this publication, and will not be made available for other purposes or to third parties.

MAKE A SUBMISSION

LANGUAGE

English

Português (Brasil)

Español (España)

Appendix D

Research Ethics Committee Approvals

This appendix presents the approvals (in Brazilian Portuguese) of all elaborated final reports of the three research projects sent to the Ethics Research Committee affiliated with Universidade Estadual de Campinas.

PARECER CONSUBSTANCIADO DO CEP

DADOS DO PROJETO DE PESQUISA

Título da Pesquisa: Análise de Problemas de Programação em Disciplinas de Introdução à Programação em Python

Pesquisador: ERYCK PEDRO DA SILVA

Área Temática:

Versão: 2

CAAE: 51444121.5.0000.5404

Instituição Proponente: Instituto de Computação

Patrocinador Principal: CONSELHO NACIONAL DE DESENVOLVIMENTO CIENTIFICO E TECNOLOGICO-CNPQ

DADOS DA NOTIFICAÇÃO

Tipo de Notificação: Envio de Relatório Final

Detalhe:

Justificativa: Esta notificação está sendo enviada como relatório final, informando a conclusão da

Data do Envio: 12/12/2022

Situação da Notificação: Parecer Consubstanciado Emitido

DADOS DO PARECER

Número do Parecer: 5.820.142

Apresentação da Notificação:

Pesquisadores enviam relatório final de atividades do projeto citado acima

Objetivo da Notificação:

Apresentar relatório final de atividades do estudo

Avaliação dos Riscos e Benefícios:

Mantidos em relação ao projeto original

Comentários e Considerações sobre a Notificação:

- Data da aprovação do projeto por este CEP: 24/11/2021 (parecer número 5.124.692, em 'PB_PARECER_CONSUBSTANCIADO_CEP_5124692.pdf', de 24/11/2021 16:22:32)

Endereço: Rua Tessália Vieira de Camargo, 126, 1° andar do Prédio I da Faculdade de Ciências Médicas
Bairro: Barão Geraldo **CEP:** 13.083-887
UF: SP **Município:** CAMPINAS
Telefone: (19)3521-8936 **Fax:** (19)3521-7187 **E-mail:** cep@unicamp.br

- Data de conclusão do estudo: 06/12/2022

Embora tenha sido, originalmente, previsto 100 voluntários para o projeto, foram incluídos 32 participantes nesse estudo
Não houve registro de intercorrências

Informa o pesquisador que "Não ocorreram mudanças na metodologia tampouco no público-alvo definidos originalmente" e que "Todos os protocolos foram seguidos conforme a descrição no projeto de pesquisa aprovado"

Não houve publicação dos resultados
Sobre isso, o pesquisador acrescenta que "Os resultados estão sendo compilados no formato de um Relatório Técnico, a ser publicado nos meios de comunicação da UNICAMP."

Considerações sobre os Termos de apresentação obrigatória:
Para avaliação desta notificação foi analisado o relatório final anexado no documento intitulado 'NOTIFICACAO_CONCLUSAO.pdf', de 12/12/2022 15:36:13

Relatório enviado adequadamente, em formulário próprio deste CEP

Conclusões ou Pendências e Lista de Inadequações:
Relatório final aprovado

Este parecer foi elaborado baseado nos documentos abaixo relacionados:

Tipo Documento	Arquivo	Postagem	Autor	Situação
Envio de Relatório Final	NOTIFICACAO_CONCLUSAO.pdf	12/12/2022 15:36:13	ERYCK PEDRO DA SILVA	Postado

Situação do Parecer:
Aprovado

Necessita Apreciação da CONEP:
Não

CAMPINAS, 15 de Dezembro de 2022

Assinado por:
Renata Maria dos Santos Celeghini
(Coordenador(a))

Endereço: Rua Tessália Vieira de Camargo, 126, 1° andar do Prédio I da Faculdade de Ciências Médicas
Bairro: Barão Geraldo **CEP:** 13.083-887
UF: SP **Município:** CAMPINAS
Telefone: (19)3521-8936 **Fax:** (19)3521-7187 **E-mail:** cep@unicamp.br

UNIVERSIDADE ESTADUAL DE CAMPINAS - UNICAMP/CAMPUS CAMPINAS

PARECER CONSUBSTANCIADO DO CEP

DADOS DO PROJETO DE PESQUISA

Título da Pesquisa: Aplicação de uma Observação Semiestruturada em Turma de Introdução à Programação

Pesquisador: ERYCK PEDRO DA SILVA

Área Temática:

Versão: 3

CAAE: 60258622.8.0000.5404

Instituição Proponente: Instituto de Computação

Patrocinador Principal: Financiamento Próprio

DADOS DA NOTIFICAÇÃO

Tipo de Notificação: Envio de Relatório Final

Detalhe:

Justificativa: Esta notificação está sendo enviada como relatório final, informando a conclusão da

Data do Envio: 20/07/2023

Situação da Notificação: Parecer Consubstanciado Emitido

DADOS DO PARECER

Número do Parecer: 6.297.763

Apresentação da Notificação:

As informações contidas nos campos "Apresentação da Notificação", "Objetivo da Notificação" e "Avaliação dos Riscos e Benefícios" foram obtidas dos documentos apresentados para apreciação ética e das informações inseridas pelo Pesquisador Responsável do estudo na Plataforma Brasil.

Foi encaminhada a notificação para análise do CEP do relatório final da pesquisa.

Objetivo da Notificação:

Encerramento do estudo no Brasil (envio de relatório final) – quando todas as etapas previstas foram finalizadas.

Avaliação dos Riscos e Benefícios:

Mantidos em relação à última versão do projeto aprovado.

Comentários e Considerações sobre a Notificação:

Segundo informações contempladas no relatório final, a pesquisa foi concluída em 17/7/2023.

Foram incluídos 29 participantes de pesquisa do total de 182 previstos na folha de rosto.

Não houve intercorrências com os participantes de pesquisa.

De acordo com os comentários do pesquisador responsável contemplados nos itens 10 e 11 do relatório:

"Esta notificação está sendo enviada como relatório final, indicando a conclusão da pesquisa. O relatório aqui apresentado contempla a Emenda aprovada pelo CEP em fevereiro de 2023. Os números de participantes informados correspondem às duas turmas de Algoritmos e Programação de Computadores (MC102) abordadas (uma no segundo semestre letivo de 2022 e a outra no primeiro semestre letivo de 2023). Não ocorreram mudanças nas metodologias propostas na Emenda. Todos os protocolos foram seguidos conforme descritos no projeto aprovado."

"Embora os resultados ainda não tenham sido publicados, um Artigo Completo contendo os resultados foi elaborado e submetido para a Revista Brasileira de Informática na Educação em julho de 2023. Até o momento do envio deste relatório ao CEP, o trabalho submetido se encontra em avaliação. Link para o site da revista: https://sol.sbc.org.br/journals/index.php/rbie/index"

Considerações sobre os Termos de apresentação obrigatória:

Vide campo abaixo "Conclusões ou Pendências e Lista de Inadequações"

Conclusões ou Pendências e Lista de Inadequações:

Diante do exposto, o Comitê de Ética em Pesquisa - CEP, de acordo com as atribuições definidas na Resolução CNS nº 466 de 2012 e na Norma Operacional nº 001 de 2013 do CNS, manifesta-se pela aprovação da notificação apresentada para o projeto de pesquisa.

Considerações Finais a critério do CEP:

Este parecer foi elaborado baseado nos documentos abaixo relacionados:

Tipo Documento	Arquivo	Postagem	Autor	Situação

Envio de Relatório Final	NOTIFICACAO_CONCLUSAO.docx	20/07/2023 12:03:04	ERYCK PEDRO DA SILVA	Postado
Envio de Relatório Final	NOTIFICACAO_CONCLUSAO.pdf	20/07/2023 12:03:07	ERYCK PEDRO DA SILVA	Postado

Situação do Parecer:
Aprovado

Necessita Apreciação da CONEP:
Não

CAMPINAS, 13 de Setembro de 2023

Assinado por:
Renata Maria dos Santos Celeghini
(Coordenador(a))

UNIVERSIDADE ESTADUAL DE CAMPINAS - UNICAMP/CAMPUS CAMPINAS

PARECER CONSUBSTANCIADO DO CEP

DADOS DO PROJETO DE PESQUISA

Título da Pesquisa: Aplicação e Avaliação de Artefatos Educacionais em Turma de Introdução à Programação

Pesquisador: ERYCK PEDRO DA SILVA

Área Temática:

Versão: 2

CAAE: 70220523.0.0000.5404

Instituição Proponente: Instituto de Computação

Patrocinador Principal: Financiamento Próprio

DADOS DA NOTIFICAÇÃO

Tipo de Notificação: Envio de Relatório Final

Detalhe:

Justificativa: Esta notificação está sendo enviada como relatório final, indicando a conclusão da

Data do Envio: 15/02/2024

Situação da Notificação: Parecer Consubstanciado Emitido

DADOS DO PARECER

Número do Parecer: 6.713.604

Apresentação da Notificação:

Pesquisadores enviam relatório final de atividades do projeto citado acima

Objetivo da Notificação:

Apresentar relatório final de atividades do estudo

Avaliação dos Riscos e Benefícios:

Mantidos em relação ao projeto original

Comentários e Considerações sobre a Notificação:

Data da aprovação do projeto por este CEP: 13/07/2023 (parecer número 6.180.304, em 'PB_PARECER_CONSUBSTANCIADO_CEP_6180304.pdf', de 13/07/2023 10:27:40)

Data de conclusão do estudo: 22/01/2024

Endereço: Rua Tessália Vieira de Camargo, 126, 1° andar do Prédio I da Faculdade de Ciências Médicas
Bairro: Barão Geraldo **CEP:** 13.083-887
UF: SP **Município:** CAMPINAS
Telefone: (19)3521-8936 **Fax:** (19)3521-7187 **E-mail:** cep@unicamp.br

Embora originalmente previstos 90 voluntários para essa pesquisa, foram incluídos 27 participantes no estudo.

Não houve registro de intercorrências
Não houve publicação dos resultados

Informa também o pesquisador que
- "Não ocorreram mudanças nas metodologias propostas. Todos os protocolos foram seguidos conforme descritos no projeto aprovado"
- "Os artefatos educacionais produzidos na pesquisa foram consolidados e disponibilizados para uso por docentes e discentes de introdução à programação"

Considerações sobre os Termos de apresentação obrigatória:
Para avaliação desta notificação foi analisado o relatório final anexado no documento intitulado 'NOTIFICACAO_CONCLUSAO.pdf', de 15/02/2024 18:44:44

Relatório enviado adequadamente, em formulário próprio deste CEP

Recomendações:
(nenhuma)

Conclusões ou Pendências e Lista de Inadequações:
Relatório final aprovado

Este parecer foi elaborado baseado nos documentos abaixo relacionados:

Tipo Documento	Arquivo	Postagem	Autor	Situação
Envio de Relatório Final	NOTIFICACAO_CONCLUSAO.pdf	15/02/2024 18:44:44	ERYCK PEDRO DA SILVA	Postado

Situação do Parecer:
Aprovado

Necessita Apreciação da CONEP:
Não

CAMPINAS, 20 de Março de 2024

Assinado por:
Renata Maria dos Santos Celeghini
(Coordenador(a))

Endereço: Rua Tessália Vieira de Camargo, 126, 1° andar do Prédio I da Faculdade de Ciências Médicas
Bairro: Barão Geraldo **CEP:** 13.083-887
UF: SP **Município:** CAMPINAS
Telefone: (19)3521-8936 **Fax:** (19)3521-7187 **E-mail:** cep@unicamp.br