Analyzing the Impact of Programming Mistakes on Students' Programming Abilities

Antonio D. dos S. Júnior[1,2], **Jorge Figueiredo**[1], **Dalton Serey**[1]

[1]Software Practices Laboratory – Universidade Federal de Campina Grande (UFCG)
Rua Aprígio Veloso, 882 – 58429-900 – Campina Grande – PB – Brazil

[2]Instituto Federal da Paraíba - Campus Picuí
Acesso à Rodovia PB 151, s/n – 58187-000 – Picuí – PB – Brazil

antonio.dias@ifpb.edu.br, abrantes@dsc.ufcg.edu.br, dalton@dsc.ufcg.edu.br

Abstract. *Despite all the research on students misconceptions in introductory programming courses, there is still a lot to understand about how they make and correct mistakes. The reviewed works investigate code as a final, static artifact, but this does not tell the whole story about the students' mistakes. With this study, we take the first step toward an attempt to close this gap. By capturing granular states of the coding process, we collected and analyzed data in an experiment with twenty-two novice students aged 14 to 16. With the experiment, we found the most common issues in the students' code. This enabled us to know which mistakes may prevent them from reaching a correct solution, and which will not interfere but may show an understanding problem in core programming concepts. This experiment precedes the development of an approach to help the teacher or teaching assistant in providing personalized and directed help to the students in programming courses.*

1. Introduction

Mistakes made by novice programmers have been researched by studies on computer science education for decades [Spohrer and Soloway 1986, Denny et al. 2012, Hristova et al. 2003]. Yet, there is still much to understand about how students make and correct certain mistakes. The investigation of student errors still present interesting research challenges. In part of our reviewed literature [De Oliveira et al. 2013, Altadmri and Brown 2015, Bulmer et al. 2018], the artifact used to check for programming mistakes is usually the last submitted code or intermediary versions that lead to the final code. Other approaches, such as analysis of the compiler feedback might reveal enough information to help solve a problem in the code, although it may not truly reflect the underlying issue [McCall and Kölling 2014, Kohn 2019].

These artifacts, however, do not tell the whole story about why and how students made certain mistakes and how they proceeded to correct them. They just show either the final solution or sparse iteration states of the code buildup, while much of the student's thought process happens in between those states. Analyzing all the steps that students take to write a piece of code may reveal a lot about the difficulties they face. This study aimed to contribute to the understanding of programming mistakes and learning difficulties. To begin to understand that, we analyzed intermediary states of code written by students in an introductory programming course.

In this experiment, twenty-two students novice students aged 14 to 16 wrote solutions for 6 programming problems to test their knowledge on variables, types, expressions, input and output functions. To collect their code, we developed a Web platform through which it is possible for the teacher to create activities containing one or more programming problems. The problems that can currently be solved in Python 2.7 and contain a title, description and a list of tests. Each test is a pair of expected input and output. The students enter their code on a Web text editor and can verify if their code passes the tests. The justification to write yet another educational coding platform is that we needed a different way to gather data. Our application must record every character insertion and removal, so we can capture a more granular progress of the code-writing process, and not only the final code.

By manually checking code the students wrote before the experiment, we found 31 types of issues and later developed a code analyzer to automatically identify them in students' code. The data we collected in our experiment was submitted to the code analyzer, and 15 out of 31 identifiable issues were found in the data set. We summarized the most common mistakes and issues related to the the way the students write code. We also investigated how the mistakes relate to correct or incorrect solutions.

The analyzer we developed is different from a linter, such as *pylint*, because our concern is more focused into identifying errors committed by students in introductory programming courses, rather than verifying formatting discrepancy, adherence to coding standards or some of the suspicious constructs usually flagged by a linter. We may inspect a number of problems that are already identifiable by a linter, but the non-overlapping issues checked by our system are more significant to our study.

Earlier studies show that programming is a subject in which knowledge must be built on previous learning. Acquiring one concept makes understanding other linked topics easier, while failing makes it harder [Qian and Lehman 2017, Basnet et al. 2018]. One of the concepts that should be solid for programming students since the beginning is the language syntax. However, it has been a long-lasting problem in introductory programming courses. Many novices find it frustrating to deal with syntax, semantic and logical errors [Denny et al. 2012, Hristova et al. 2003, Smith and Rixner 2019, Spacco et al. 2015]. With the knowledge acquired in this research, we intend to develop an approach in a future study to facilitate the identification of issues in students' code, making it easier to understand their learning difficulties. We aim to help teachers and teaching assistants to earlier identify issues in code. This would allow them to understand and tackle the students' programming issues sooner, and thus help them improve their comprehension by dealing with their learning challenges.

This paper is organized as follows. A review of works related to ours is presented in Section 2. In Section 3, we describe details of the system we built, how we used it to collect data and explain what comprises our data set. Section 4 shows the results interpretation and final remarks are presented in Section 5.

2. Related Work

Novice programmers' mistakes cannot be placed under a single category. Researches use distinct methods to investigate errors under several points of view.

In [Liu and Petersen 2019], the authors developed PyTA, a static analysis tool for

Python. The tool identifies common CS1 student errors and provides intuitive error messages to aid them in debugging. The authors state that compiler error messages are difficult to use. For the students, static analysis can provide additional information that can be used to debug programs more effectively. Likewise, static analysis can provide some insight into students difficulties to the teacher.

Kohn [Kohn 2019] wrote a parser that detects and identifies over one hundred different syntax errors. One of the author's goals was to develop a metric that measures how well compiler error messages fit the underlying errors and if the messages enable students to correct their code. According to the author, the goals and plans behind a student's program must be known. Otherwise, the nature of certain errors that may indicate misconceptions or problems cannot always be reliably determined, even by humans.

Smith and Rixner [Smith and Rixner 2019] provided a characterization of errors that occurred in over 330,000 Python implementations of eight functions by novice programmers. The research presented a quantitative analysis of the distribution, duration, and evolution of the errors made in the code. They addressed temporal and relational questions. Among them were: which errors students may encounter earlier in the development process and which pairs of errors students frequently transition between while working towards their final solution.

Hristova *et al.* [Hristova et al. 2003] made a survey to collect a list of Java Programming errors reported by teacher, teaching assistants and students. A total of 20 errors were considered in the development of the tool – a multi-pass pre-processor. This tool translates the compiler messages into friendlier comments.

Another research explored a different aspect of syntax errors. Denny *et al.* [Denny et al. 2012] investigated syntax errors made by students when writing short code fragments in Java, and the time they spent correcting them. They concluded that "cannot resolve identifier" was the most common error found in the code of 356 out of 385 students. That error, in addition to "type mismatch", present in 341 code samples, were the errors that dominated students' time. One interesting discovery was that regardless of whether their ability to code, all students spent almost the same amount of time correcting the common errors.

Three similar works investigated the most common syntax and semantic errors under different contexts. Altadmri and Brown [Altadmri and Brown 2015] analyzed a set of compilation events from 250,000 students all over the world. The data was extracted from the Black-box data set, which collects Java code written by users of BlueJ, a Java IDE for beginners. The authors observed what are the most frequent mistakes in a large-scale multi-institution data set; the most common errors, and common classes of errors; which errors take the shortest or longest time to fix, and how do these errors evolve during the academic terms and academic year. Ettles et al. [Ettles et al. 2018] researched the most common logic errors that students make, and which are the most problematic for them to identify and fix. The most common errors were: *integer division*, when students failed to realize that the division of two integers evaluates to an integer; *uninitialized variables*; and *off by one error*. In a similar study to ours, [Bulmer et al. 2018] examined coding habits of 77 first-year students. The authors developed a tool to automatically detect relevant code patterns and visualize the coding process in real time. A subset of commonly known syntax and semantic errors, and a selection of poor coding practices were chosen by the authors. Among the bad habits detected by the study, the most common ones were found to be *unclosed scanners* and *brackets and quotes miscounts*.

In [De Oliveira et al. 2013], the authors developed a system to recommend classes of activities to students based on their profile. This recommendation system maps profiles which, in a similar way to our work, include the students' learning difficulties. While we do not directly mount a student profile, the issues we capture in students' code could compose a profile that maps their difficulties. This would hold relevant information to show how their learning could be improved.

3. Methodology

3.1. Context

The experiment was carried out at a first-year programming course at Federal Institute of Paraíba, campus Picuí in 2018. Twenty-two students aged 14 to 16 from the high-school integrated to vocational training course in informatics participated in this research. All students were enrolled in an introductory algorithms and programming logic course, compulsory to the informatics students. The students and their respective legal guardians signed an informed consent form authorizing the use of their anonymous data in the research.

To pursue the experiment, we built a Web platform that made possible the continuous capture of data inputted into a text editor by the students. The Web platform architecture consisted of a front-end built in Angular.js [1] that communicated to a Web API developed in Python with Flask [2].

The system built was used for an entire school year, but we developed a code analyzer module specifically for the study, and we plan to permanently integrate it into the main system. The module was integrated into the Web API and provided analysis of stored code previously captured. This module works in two fronts: it searches for patterns of issues in the code by traversing its abstract syntax tree, and generates a trace to capture how the variables values change throughout the program execution. Knowing the variables values and how they change is useful to find certain problems in code. Currently, our analyzer can only evaluate syntactically valid code, but we are working on another part capable of finding common issues in syntactically invalid code.

By manually checking code previously written by the students, we found a total of 31 issues that could be summarized as patterns. We then coded the analyzer module to automatically find these patterns. When we submitted the experiment data set to the analyzer, 15 out of the 31 identifiable issues were found. Due to space restrictions, we detailed only the 15 found issues below.

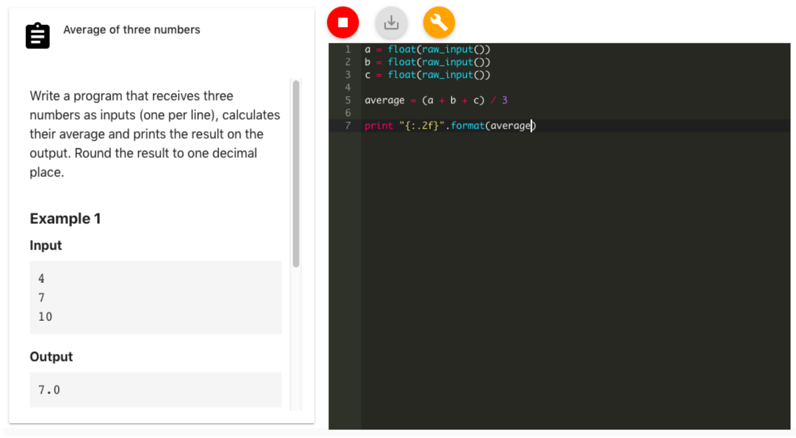
1. **Integer division:** indicates that an integer division was used when it probably should have been a float division, causing loss of precision.
2. **Invalid addition:** represents an incorrect use of the '+' operator, such as applying it to an integer and a string.

[1]https://angularjs.org/
[2]https://palletsprojects.com/p/flask/

3. **Redundant division:** shows that a division by one was used. While this is not an issue in the code, it may indicate a math understanding gap.
4. **Redundant multiplication:** shows that a multiplication by one was used. This is similar to issue 3.
5. **Redundant float conversion:** indicates that a float conversion was done on a literal or variable that was already a float.
6. **Redundant int conversion:** indicates that an integer conversion was done on a literal or variable that was already an int.
7. **Should use int():** flags that the function int() should have been used in the code. Our tool checks the code for the presence of a list of functions defined by the teacher or assistant.
8. **Should use float():** same as issue 7.
9. **Type change:** indicates that a variable's type has changed at some point in the code.
10. **Unassigned expression:** flags that an expression appears on one line and its result is never assigned to a variable.
11. **Undefined value:** flags when an invalid value is used in an expression.
12. **Undefined variables:** indicates that a undeclared variable has been used.
13. **Unused variables:** indicates that a variable is declared, but never used.
14. **Wrong input reading:** shows that an incorrect input reading was performed.
15. **Wrong output reading:** shows that an incorrect output writing was performed.

3.2. Data Collection

Students used the Web platform that we developed to solve six programming problems. The programming problems involved the use of simple mathematical expressions and operations (addition, subtraction, division, multiplication and modulo), and tested the students' knowledge of variables, types, expressions, input and output functions. A testing functionality allowed the participant to know if their code passed the predefined tests. All solutions were submitted by the end of the experiment's deadline. One example of a problem, how the tool presents it and the text editor are shown in Figure 1.



Average of three numbers

Write a program that receives three numbers as inputs (one per line), calculates their average and prints the result on the output. Round the result to one decimal place.

Example 1

Input

```
4
7
10
```

Output

```
7.0
```

```
a = float(raw_input())
b = float(raw_input())
c = float(raw_input())

average = (a + b + c) / 3

print "{:.2f}".format(average)
```

Figure 1. The problem description and text editor

The collected data for each problem solution was a list of entities representing events of text insertion, deletion or selection, the respective timestamp and location (line and column in the editor) associated with each event. This allowed us later to have a dynamic view of the student's coding process, rather than a static, finished code, that sometimes would not provide the information we were looking for.

We can use this data to reconstruct the whole code that the student wrote, character-by-character. An example of the list of data captured is given in Listing 1. In the example, 12 insertion events represent 12 characters written by the student (the editor automatically closed the parenthesis on the 12th event), corresponding to the text `a=raw_input()`.

Listing 1. Example of insertion events collected by the tool

```
{ac: "i", e: [0,1], ls: ["a"], sn: 0, ts: "2019-06-24T11:29:00.724Z"},
{ac: "i", e: [0,2], ls: ["="], sn: 1, ts: "2019-06-24T11:29:04.776Z"},
{ac: "i", e: [0,3], ls: ["r"], sn: 2, ts: "2019-06-24T11:29:06.304Z"},
{ac: "i", e: [0,4], ls: ["a"], sn: 3, ts: "2019-06-24T11:29:06.502Z"},
{ac: "i", e: [0,5], ls: ["w"], sn: 4, ts: "2019-06-24T11:29:06.629Z"},
{ac: "i", e: [0,6], ls: ["_"], sn: 5, ts: "2019-06-24T11:29:06.942Z"},
{ac: "i", e: [0,7], ls: ["i"], sn: 6, ts: "2019-06-24T11:29:07.977Z"},
{ac: "i", e: [0,8], ls: ["n"], sn: 7, ts: "2019-06-24T11:29:08.051Z"},
{ac: "i", e: [0,9], ls: ["p"], sn: 8, ts: "2019-06-24T11:29:08.266Z"},
{ac: "i", e: [0,10], ls: ["u"], sn: 9, ts: "2019-06-24T11:29:08.470Z"},
{ac: "i", e: [0,11], ls: ["t"], sn: 10, ts: "2019-06-24T11:29:08.595Z"},
{ac: "i", e: [0,13], ls: ["()"], sn: 11, ts: "2019-06-24T11:29:09.093Z"}
```

4. Results and Discussion

The number of correct and wrong submissions for each problem is shown in Figure 2. A submission is accepted as correct only when it passes all the test cases associated with it. The absence of a submission also counts as wrong. Using the ratio of correct and wrong submissions as the difficulty level of a problem, "Employee of the month" and "Savior" are the most difficult problems.

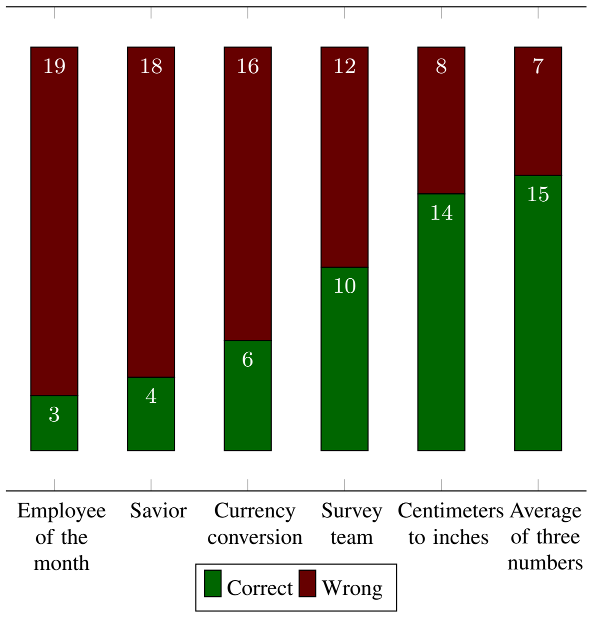


Figure 2. Number of correct and wrong submissions

The number of problems solved by students is shown in Figure 3. This chart helps us have an idea of the students performance level. Out of the 22 students that participated on the experiment, 12 solved 2 problems or less. Among the 12 students that solved 2 problems or less, 8 students solved the problems “Centimeters to inches” and/or “Average of three numbers”, 3 students did not submit any correct answer, and 1 of them had correct submissions for the problems “Survey team” and “Average of three numbers”.

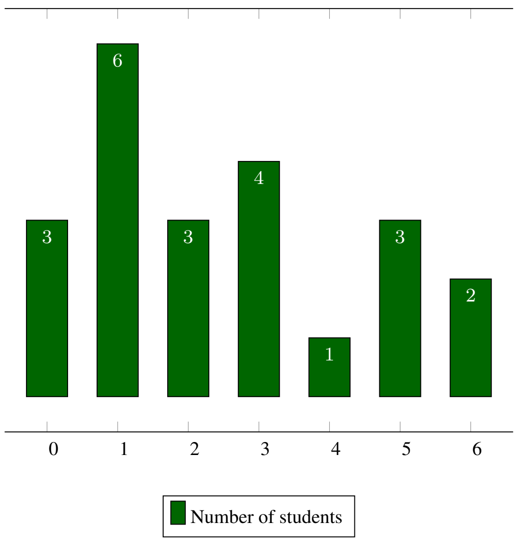


Figure 3. Number of problems solved by students

The top ten issues, along with the number of their occurrences found for each category, are shown in Figure 4. An issue is considered to be present in the code when it appeared in one of the syntactically correct states of code and wasn’t corrected. The most frequent issues present in code samples analyzed were: *unused variables*, *redundant float conversion* and *wrong input reading*. The less frequent ones, which do not appear in the chart, are *type change* (2 occurrences), *invalid addition*, *unassigned expression*, *redundant int conversion* and *redundant multiplication*, each with one occurrence. Only one occurrence is counted for each problem solved by a student.

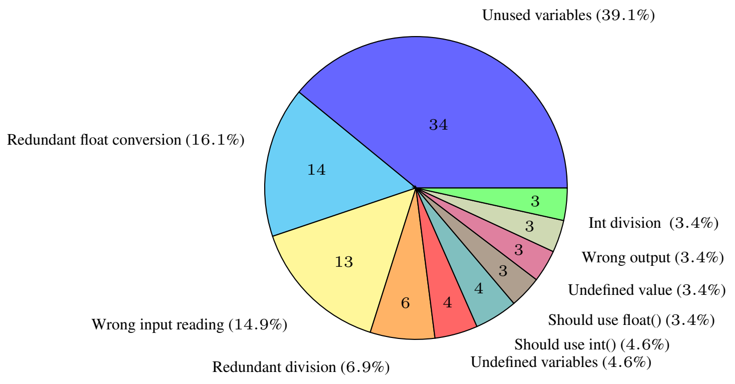


Figure 4. Top ten issues found by category and number of occurrences

The occurrence of *redundant float conversion* may indicate a poor understanding of types, and *wrong input reading* could mean that the student did not understand the problem description correctly. But it is hard to tell why they leave unused variables in the code.

There were issues that appeared most frequently in incorrect submissions. *Unused variables*, with 31 occurrences, *wrong input reading*, with 13 occurrences and *undefined variables*, *redundant float conversion* and *should use* `int()`, each with 4 occurrences. The most frequent issues in wrong submissions are shown in Figure 5.

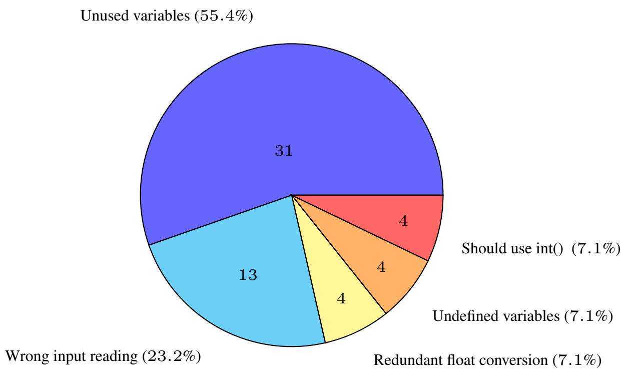


Figure 5. Most frequent issues in wrong submissions

The higher occurrence of certain problems in incorrect submissions may indicate that these issues prevent a student from writing a correct solution for a problem. This fact may simply show that the students did not comprehend the problem description. This makes them read the inputs incorrectly, and leave unused variables in the code, when they quit trying to solve the problem. The presence of undefined variables in code could mean that the student did not understand that a variable must exist before they use operators such as +=, or that they just did not initialize it.

Unlike the occurrence of *unused variables*, the other issues (*wrong input reading* and *should use* `int()`) may cause runtime errors under some or all test cases. These issues, along with *redundant float conversion*, may indicate that the student has a poor understanding of types or does not know how to approach the solution to the problem.

On the other hand, some issues appeared mostly on correct submissions: *redundant float conversion* occurred 10 times and redundant division and unused variables each appeared 3 times, as shown in Figure 6. This seems to indicate that these issues do not pose a barrier for the student to reach a correct solution.

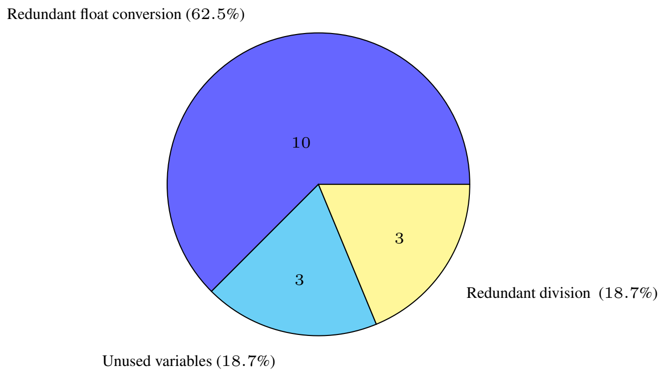


Figure 6. Most frequent issues in correct submissions

The issues *redundant float conversion*, *redundant division* and *unused variables* do not present a serious setback in the student's development of the problem. These programming mistakes cause less impact in the solution development. They are just redundant code (and computation) that the student may have forgotten to clean up, causing no errors on runtime.

The results showed interesting information that, if given to a teacher or teaching assistant, will allow them to provide efficient help to the students. While that is a good point, it may not be precise enough information. It is hard to tell the reason why students make certain mistakes without getting their feedback. In that case, using different approach in an experiment, such as the think-aloud protocol [Kadekar et al. 2018] would help us getting a more precise view of the problem the student is facing.

5. Conclusion

Many times, the student will not communicate their difficulties, or when they do, they cannot clearly point out what their exact problem is. The results of this study showed that we are able to search code for issues and potential problems. These may indicate a gap in the student's understanding of programming concepts. We may be able to point out why a student is reaching a correct solution despite of some lack of understanding, or why some issues stop them from reaching a correct solution.

With the knowledge obtained in this study, we want to prepare a new study to get more accurate answers on why students make certain programming mistakes. This study was also the precursor of a work to develop and validate a diagnostic approach to ease the teacher's job on identifying students' difficulties. A diagnostic tool may help the teacher decide where to focus their attention when preparing after-class help plans, providing personalized help or new programming problems, for example.

References

Altadmri, A. and Brown, N. C. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 522–527. ACM.

Basnet, R., Payne, L., Doleck, T., Lemay, D., and Bazelais, P. (2018). Exploring bimodality in introductory computer science performance distributions. *Eurasia Journal of Mathematics, Science and Technology Education*, 14.

Bulmer, J., Pinchbeck, A., and Hui, B. (2018). Visualizing code patterns in novice programmers. In *Proceedings of the 23rd Western Canadian Conference on Computing Education*, page 7. ACM.

De Oliveira, M. G., Ciarelli, P. M., and Oliveira, E. (2013). Recommendation of programming activities by multi-label classification for a formative assessment of students. *Expert Systems with Applications*, 40(16):6641–6651.

Denny, P., Luxton-Reilly, A., and Tempero, E. (2012). All syntax errors are not equal. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, pages 75–80. ACM.

Ettles, A., Luxton-Reilly, A., and Denny, P. (2018). Common logic errors made by novice programmers. In *Proceedings of the 20th Australasian Computing Education Conference*, pages 83–89. ACM.

Hristova, M., Misra, A., Rutter, M., and Mercuri, R. (2003). Identifying and correcting java programming errors for introductory computer science students. In *ACM SIGCSE Bulletin*, volume 35, pages 153–156. ACM.

Kadekar, H. B. M., Sohoni, S., and Craig, S. D. (2018). Effects of error messages on students' ability to understand and fix programming errors. In *2018 IEEE Frontiers in Education Conference (FIE)*, pages 1–8.

Kohn, T. (2019). The error behind the message: Finding the cause of error messages in python. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 524–530, New York, NY, USA. ACM.

Liu, D. and Petersen, A. (2019). Static analyses in python programming courses. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 666–671, New York, NY, USA. ACM.

McCall, D. and Kölling, M. (2014). Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–8. IEEE.

Qian, Y. and Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Educ.*, 18(1):1:1–1:24.

Smith, R. and Rixner, S. (2019). The error landscape: Characterizing the mistakes of novice programmers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 538–544, New York, NY, USA. ACM.

Spacco, J., Denny, P., Richards, B., Babcock, D., Hovemeyer, D., Moscola, J., and Duvall, R. (2015). Analyzing student work patterns using programming exercise data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 18–23, New York, NY, USA. ACM.

Spohrer, J. C. and Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? *Commun. ACM*, 29(7):624–632.